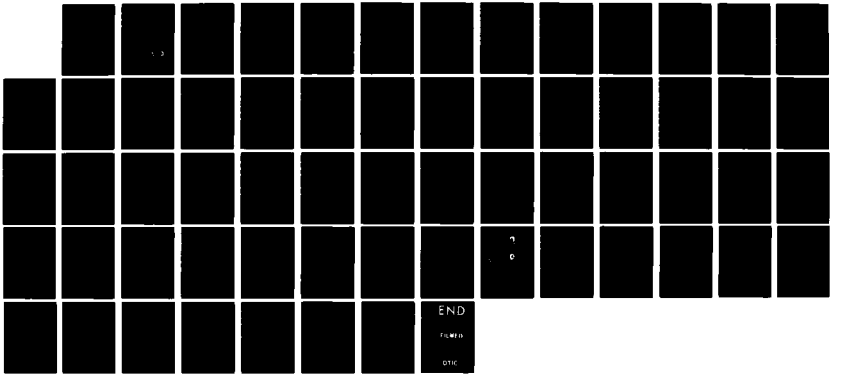
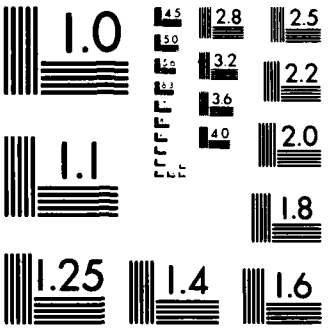
AD-A149 140 RESEARCH INTO SELF-TIMED VLSI CIRCUITS(U) PRINCETON UNIV NJ DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON 22 OCT 84 N00014-82-K-0549 1/1

UNCLASSIFIED F/G 9/5 NL





(12)

PRINCETON VLSI PROJECT: Semi-Annual Report

PERIOD ENDING: October 22, 1984

Richard J. Lipton -- Principal Investigator

EECS Department
PRINCETON UNIVERSITY

Title: Research into Self-Timed VLSI Circuits
Contract N00014-82-K-0549

FACULTY:

Bruce W. Arden, Chairman
David P. Dobkin
Hector Garcia-Molina
Peter Honeyman
Andrea LaPaugh
Kenneth Steiglitz
Kenneth Supowit

Princeton VLSI Project

B. Arden, D. Dobkin, H. Garcia-Molina, P. Honeyman
A. LaPaugh, R. Lipton, K. Steiglitz, K. Supowit

1. Introduction

There are three major components to our project. The first is in the area of procedural design of VLSI circuits. The second is our census language and techniques, and the third is in the area of the testing of VLSI circuits.

2. Procedural Approach to VLSI Design

2.1. ALLENDE [LaPaugh, Mata, Heng, Lin, Yeh]

ALLENDE is a new language for VLSI design based on our earlier work on ALI and Clay. Several new ideas have been introduced to make it both easier to use and more efficient. First, a layout is constructed in a structured way. Second, wires which were explicit in our earlier languages are now implicit. This eliminates the need for tedious naming of wires and resulting errors. Additionally, the structured nature of ALLENDE forces all design rule violations to be caught by the system; hence, one need not use a standard design rule checker.

Internally, ALLENDE generates not CIF but a higher level form which we call PIF. PIF also is a useful tool for interfacing other design tools. We are currently using both ALLENDE and PIF to build a variety of other tools. Lin has written a channel router based on the Rivest-Fiduccia algorithm. Heng has written a pad router; he is now making it work with the MOSIS pad frames. The Berkeley PLA generator has been interfaced to ALLENDE; we are now building a Weinberger array generator.

2.2. Clay [North]

The Clay procedural layout system is the primary design tool for the Princeton Reduced Instruction Set Machine (PRISM) project. The control path bitslice has been fabricated and is in test. The data path chip will be submitted to MOSIS for fabrication shortly. These projects have given us experience with Clay in creating large VLSI layouts. The Clay system has helped to shape the ALLENDE design language, and has also given us insight into desirable characteristics for CAD tools combining both procedural code and graphics and efficient techniques for implementing procedural layout systems.

2.3. Applications [Steiglitz]

ALLENDE is being used in the design phase of a project to study cellular automata. The project is examining the capabilities of cellular automata as a model of general nonlinear phenomena. The implementation of cellular automata as VLSI chips will allow experimentation that is too time-consuming or expensive using general purpose computers. Currently, a multiprocessor cellular automaton chip with programmable next-state function is being designed. We have already fabricated and tested an 18 processor cellular automaton with a fixed next-state function; it achieves about 1.4×10^8 bit updates per second.

3. Census

3.1. Top/Down Project [Lopresti]

This project is investigating the use of the census approach to parallel computations. We currently have a four processor system running; recently four new processor boards arrived. The new boards are based on the 32-bit national chip set and include floating point and half a megabyte of memory. We are continuing our experiments on uses of the system, focusing on a variety of local search and "simulated annealing" type tasks.

3.2. ESP [Park]

A prototype version of the ESP controller hardware for use in the MMM Project is now undergoing testing. The individual ESP controllers are designed with TTL hardware and are currently on a multibus wire wrap board. These ESP controllers are being used to interconnect two Intel 8086 microprocessors together via a twenty bit wide ESP bus. The completed system will have a one megabyte memory space distributed across up to eight processors. We plan next to use the system as a test bed for a variety of issues such as synchronization, reliability, and global control.

4. Testing [LaPaugh, Steiglitz, Lucas]

We are continuing to exploit ways to use design modification to simplify testing. We are currently empirically studying a variety of methods of using additional logic at the gate level to enhance testability. The key questions are the tradeoffs between additional logic and easy of test vector generation.

5. Recent Ph.D. Theses

M. D. Huang, "Localized Graph Algorithms with Low Page-Fault Complexity".

J. M. Mata, "A Methodology for VLSI design and a Constraint-Based Layout Language".

A. S. Vergis, "Multiple Fault Detection in Digital Circuits".

6. Papers

ALLENDE: a procedural language for the hierarchical specification of VLSI layouts

José Monteiro da Mata

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

Technical Report #325
October 1984

ALLENDE: a procedural language for the hierarchical specification of VLSI layouts

José Monteiro da Mata

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

Abstract

ALLENDE is a simple and powerful layout language, associated with a structured design methodology for VLSI. It has a combination of features that set it apart from the existing VLSI layout tools. These features include the procedural language approach, the structured specification of the layout, the use of constraints to represent the layout, and the use of an intermediate form in the implementation of the system.

In ALLENDE the layout is described hierarchically as a composition of cells; absolute sizes or positions are never specified. The layout description is translated into linear constraints, which express design rules and relative position of the layout elements. By solving these constraints we obtain the absolute layout, which is guaranteed to be free of design rule violations. Errors in the layout description are immediately detected and easily located.

ALLENDE consists of five procedures to be called from a Pascal or C program, allowing the user to describe a VLSI layout. A lot of parameterization is possible when specifying layout elements, besides the ability to make use of the full power of Pascal or C. The ALLENDE layout system has been implemented for the nMOS technology. In this system we can also use cells generated by other layout tools. Our layout language can also be a target for a silicon compiler.

1. Introduction

The costs associated to the design of complex chips, the need to make integrated circuit design accessible to a larger number of people, and the need for more powerful tools to manage design changes, have forced the reevaluation of VLSI design techniques. Methods to enhance designer productivity have to be explored.

The layout phase is the most critical phase in the design of integrated circuits, because it involves expensive tools and a large amount of human intervention, and also because of its effects on

This work was supported in part by NSF Grant MCS-8004490, DARPA Contract N00014-82-K-0549, ONR Grant N00014-83-K-0275, and CAPES-Brazil.

production costs. A large part of the work in VLSI is dedicated to layout tools and techniques. The majority of the layout tools are graphics editors, like STICKS[16] and CAESAR[12]. There also exist layout languages, procedural or only descriptive, like LAVA[10], PLATES[14], HILL[5], and ALI[6][15]. Layout languages have had limited success, mainly because of being too verbose, limited in power and flexibility, and giving poor results. One of the goals of layout tools is to produce error-free layouts, but still today there is a proliferation of layout verification programs, like design rule checkers.

In this work we concentrate on the layout problem. We have two major goals in mind:

- to have a **powerful** tool that allows the designer to obtain **easily** a **correct** layout for his design;
- to have a component that can be **extended** and **integrated** with other components of a design system associated to a structured design methodology.

Our approach for VLSI layout is basically to use a **language** to hierarchically describe the circuit structure and layout topology, and to use linear **constraints** to internally represent the layout.

Our layout system, *ALLENDE*[8][9], has a combination of features that set it apart from the existing layout tools. The procedural language approach and the representation of the layout as constraints distinguish ALLENDE from all graphics-based layout editors and from most layout languages. The difference with other existing or proposed constraint-based procedural layout languages consists in the way the layout is described, the kind of constraints generated, and also the form of implementation of the system. The net result is the power, flexibility, and efficiency achieved by ALLENDE.

2. The ALLENDE Layout System

2.1. Basic Ideas

Our approach to tackle the layout problem is to have a language for the description of the layout structure. From the textual representation of the layout, constraints are generated, solved, and then the physical layout is obtained. The characteristics of the language depend, of course, on the class of objects manipulated and how they are manipulated.

In our system, the only object that we have are **cells**. A cell corresponds to a rectangle with internal structure and parameter wires.

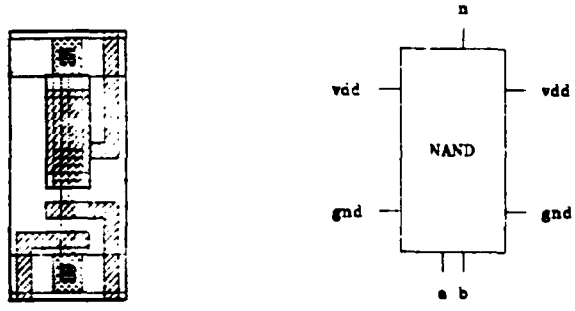


Fig. 1 - A nand cell

A composition of two cells is made by specifying their relative position (*left*, *right*, *above*, *below*), and the result is another cell. A single cell can also be rotated or flipped.

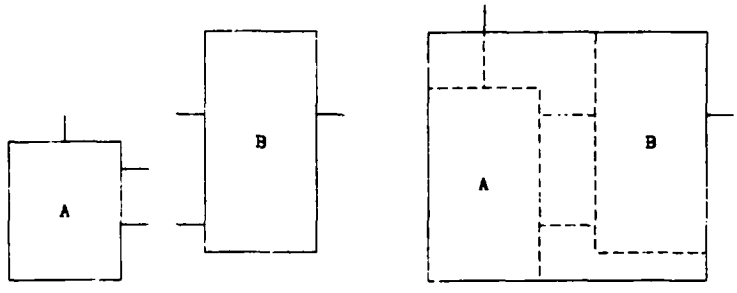
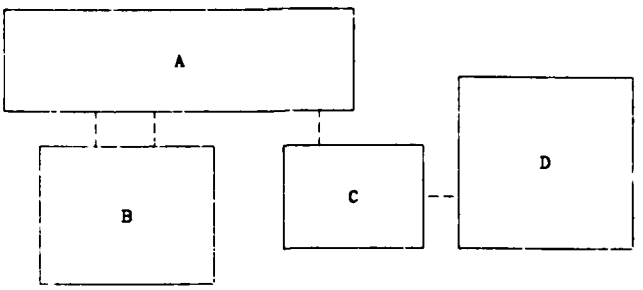


Fig. 2 - Composition of cells

When composing cells there is no need to worry about matching cell size or wire spacing (except for cells of fixed size); the conditions for matching are expressed in the linear constraints generated, and then the size of a cell will depend on the context in which it is instantiated.

The first basic idea is to describe the layout hierarchically as a composition of cells. At the bottom level of the hierarchy there will be system cells (contact, transistor, etc.) or rigid cells (previously defined layout pieces).



(A **above** (B **left** C)) **left** (**rotated90** D)

Fig. 3 - Structured layout description

When using a system cell we specify the wires on each side of the cell. When two cells are composed, the wires to be connected, and the parameter wires for the resulting cell, are determined by context.

The second basic idea is to use an intermediate language to represent the layout structure. This language should be different from the user language, but at a higher level than a mask level language, like CIF (Caltech Intermediate Form) [1]. The intent is to separate language aspects from layout aspects, or user aspects from system aspects. For layout or system aspects we mean constraint generation and layout production. For language or user aspects we mean the high level language used to describe the layout, and its implementation.

This intermediate language brings flexibility to the layout system. There may be more than one user language, even a graphics language. The implementation of the intermediate language and of the user languages are independent, and easier than the implementation without an intermediate language. The intermediate language deals with the layout structure only, while the user language may have all the power of a procedural language. The idea then is to extract all the layout information from the user program, and then process it.

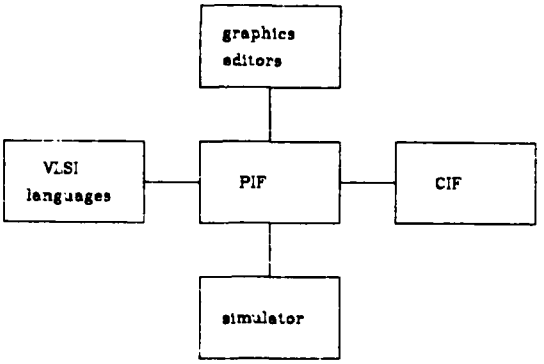


Fig. 4 - The role of the intermediate language

Based on these ideas we built a layout system. There is a user language, ALLENDE, that is no more than Pascal or C with a few procedures and functions added, and an intermediate language, PIF. The output of the user program is the layout structure in intermediate form (PIF), from which linear constraints are generated, solved, and the absolute layout in CIF is produced.

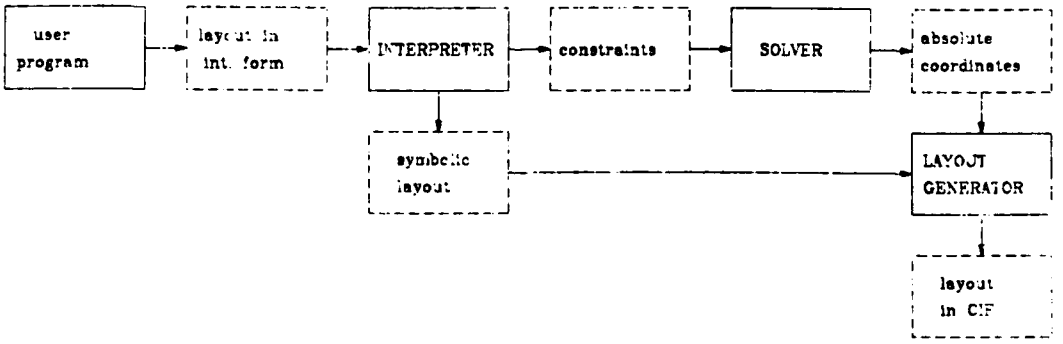


Fig. 5 - The layout generation process

This layout system works for the nMOS technology, and extensions for other technologies are under study. We use CIF to describe the final layout, although other languages could be used; in the same way, Pascal and C were chosen just for reasons of convenience. We use only right-angle geometry. The coordinate system is a half-lambda grid.

2.2. Describing Layouts Using Linear Constraints

As shown in fig. 5, in our layout system there are different representations for the layout: the user representation (in ALLENDE), the intermediate form (in PIF), the symbolic form with constraints, and the mask-level representation (in CIF). In CIF the layout objects, mainly rectangles, are described in terms of absolute coordinates; the coordinate unit is one hundredth of a micron.

Our symbolic representation of the layout is in terms of the relative coordinates of the layout elements; the relation between these coordinates is expressed by a set of linear constraints. The variables in these constraints are the X and Y coordinates of the objects in the layout. The constraints describe the interaction between the objects, and may come from the geometric design rules, connectivity, and hierarchy in the layout description. By solving the constraints and replacing the values obtained for the coordinates in the symbolic layout we obtain the absolute layout.

The set of constraints is solved in such a way to minimize the total area. Due to the large number of layout elements, the constraints ought to be as simple as possible, in order to reduce the complexity of the solving algorithm. We assume that the X and Y constraints are decoupled; this means that no constraint involves both X and Y coordinates, and that constraints involving X and Y coordinates are independent. We don't allow constraints to be related by the operator **or**, for example. By decoupling the X and Y constraints the compaction problem is made equivalent to solving two independent sets of constraints.

The whole layout is represented using constraints of the form:

$$x_i = x_j$$
$$x_i - x_j \geq d \quad (d > 0, \textit{integer})$$
$$x_i - x_j = e \quad (e > 0, \textit{integer})$$

We have an efficient algorithm to solve such constraints, described in [7]. The algorithm is based on the topological sort.

Each constraint of the form $x_i - x_j = e$ corresponds to a ***rigid*** cell. The user may control the number of constraints by constructing the layout in several steps: making rigid cells and using them at the next level of the cell hierarchy. If there are no constraints of the form $x_i - x_j = e$ in the set of constraints generated, there is always a solution to the equations, since our way of generating constraints doesn't create "cycles". The only situation when there is no solution to the set of constraints occurs when a rigid cell doesn't fit the context where it is instantiated. For example, some condition may force a larger separation between two parameter wires of a rigid cell.

2.3. The Intermediate Language PIF

The idea behind PIF is to represent the layout structure in a compact way, as in fig. 3. The objects in the layout are cells, and the operators specify position or orientation. Our layout representation is exactly like an arithmetic expression; operands are cells, binary operators specify relative position (left, right, above, below), and unary operators specify orientation (rotation or flipping). Operator precedence is as usual, and parentheses can be used to change precedence.

A layout in PIF is a structured combination of cells, while a layout in CIF is a combination of rectangles and other elements in any order. The result of the interpretation of a PIF program is a set of constraints, the layout in symbolic form (no absolute coordinates assigned), and the circuit (at the switch level) for simulation.

An example of a PIF program is given in fig. 6. The code "*C [. . d2 m3]*" represents a cell that is the contact of two wires: diffusion 2 lambda wide coming from the right and metal 3 lambda wide coming from the bottom (the two "."s indicate that no wire comes from the left or top). "**A**" means *above*, "**+**" means crossing, the parentheses delimit a cell, and "*$example*" specifies the name *example* for the cell.

```
$example
(
C [ . . d2 m3 ]
A
+ [ p2 m3 p2 m3 ]
)
```

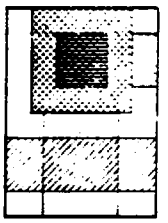


Fig. 6 - A PIF program

In PIF, layout construction is like expression evaluation. If we use a grammar to describe this layout language, the construction of the layout can be done when parsing, in a bottom-up fashion. In fact, this is similar to the way the system for typesetting mathematics EQN [4] was designed and implemented. In EQN, equations are pictured as a set of "boxes", pieced together in various ways.

Fig. 0 shows the grammar that describes the PIF language, exactly in the same format submitted to the compiler generator YACC [3]. As a PIF program is supposed to be generated by a program, and not by the user, we tried to make the language compact and easy to process, not worrying about readability, although one can read a PIF program.

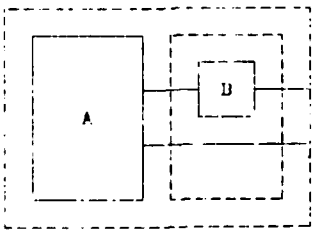
```
chip            : cell
                        { chipinterface($1); writefiles(); }
                ;
cell            : orientedcell
                | cell POSITION orientedcell
                        { $$ = compose($1,$2,$3); }
                ;
orientedcell    : singlecell
                | orientation singlecell
                        { resetorientation(); $$ = $2; }
                ;
singlecell      : syscell
                | rigidcell
                | composedcell
                | label composedcell
                        { endcell($1); $$ = $2; }
                ;
composedcell    : '(' cell ')'
                        { $$ = $2; }
                | '(' cell ')' wirenames
                        { putlabels($2,$4); $$ = $2; }
                ;
orientation     : ORIENTATION
                        { $$ = changeorientation($1); }
                ;
label           : LABEL
                        { newcell($1); $$ = $1; }
                ;
wirenames       : WIRENAMES
                        { $$ = namelist($1); }
                ;
rigidcell       : CELLNAME
                        { $$ = rigidcell($1); }
                ;
syscell         : cellcode '[' wires wires wires wires ']'
                        { $$ = syscell($1,$3,$4,$5,$6,0,0); }
                | cellcode INTEGER INTEGER '[' wires wires wires wires ']'
                        { $$ = syscell($1,$5,$6,$7,$8,$2,$3); }
                ;
cellcode        : SYSCELL
                        { $$ = cellccde = $1; }
                ;
wires           : wire
                | '|' wirelist '|'
                        { $$ = $2; }
                ;
wirelist        : wire
                | wirelist wire
                { $$ = wirelist($1,$2); }
                ;
wire            : LAYER INTEGER
                        { $$ = startwire($1,$2); }
                | '.'
                        { $$ = startwire(NOLAYER,0); }
                ;
```

Fig. 7 - PIF grammar

The lexical elements are the following:

INTEGER:	an unsigned integer;
LAYER:	*m, p, d* (for metal, polysillicon, and diffusion);
SYSCELL:	*C, X, T, I, P, +, W, J, E, N* (for contact, independent contact, transistor, implanted transistor, pullup, crossing, line, jog, empty, and nullcell, respectively);
POSITION:	*L, R, A, B* (for left, right, above, and below);
ORIENTATION:	*r0, r90, r180, r270, f0, f90, f45, f135* (for rotation and flipping);
CELLNAME:	*$cellname*
RIGIDNAME:	*&rigidname*
WIRENAMES:	*{wirenames}*
COMMENT:	*/comment/*

The smallest object that we handle is a system cell, which corresponds to a structure built according to the design rules and that forms a contact, a transistor, and so on. The cells *contact*, *transistor*, and *pullup* are the usual ones. An *implanted transistor* is a pullup with the gate not connected directly to the source. An *independent contact* represents the connection of wires of the same layer independently of other layers; it is basically used to represent independent overlapping wires in a cell. The crossing of wires in the layout has to be specified, and the cell *crossing* is used for this purpose. A *jog* represents a bend in a wire, that can move in two directions. The cell *empty* represents a cell with nothing inside, and it is useful for top-down design. The cell *nullcell* has no effect; it is like an identity element for the placement operation, and it is useful to simplify some programs that describe a layout. *line* means a single wire or a set of parallel wires; it is used in situations like the one shown in fig. 8.



A **left** (B **above** LINE)

Fig. 8 - Use of the "line" system cell

A rigid cell is a cell of fixed size; its code is in CIF, with a header giving information like size and parameter wires. *rigidname* is the name of a file containing the rigid cell. *cellname* is just the name of a cell, used mostly for debugging purposes.

Wire names are related to a cell, and they refer to the parameter wires of the cell. One of the uses of wire names is to give information for simulation.

2.4. The ALLENDE Language

ALLENDE (A Layout Language Effective for nMOS Design) is a set of procedures and functions to be called from a Pascal or C program, allowing the user to describe a VLSI layout. Basically, the user describes cells and their relative placement. Cell hierarchy comes naturally by using procedures to describe cells.

The user can make use of the full power of Pascal or C. The basic procedures and functions to describe the layout allow a great deal of parameterization, thus allowing the user to obtain completely different layouts just by changing a parameter in the program.

The output of the user program is the layout in intermediate form (PIF), from which linear constraints are generated, solved, and the absolute layout in CIF is produced. The layout obtained is guaranteed to be free of design rule violations.

The basic idea of the ALLENDE language is the same as in the PIF intermediate form: the layout is described hierarchically as a composition of cells. The difference now is that the user has available all the power of a procedural language.

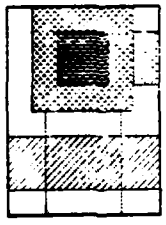
The following procedures allow the user to describe a layout:

syscell(kind,wire1,wire2,wire3,wire4,ratio)
extcell(filename)
place(operator)
begincell(cellname)
endcell(wirenames)

syscell specifies a system cell. *extcell* specifies an external cell. *begincell* and *endcell* are used to delimit a composite cell. *place* specifies the operator to be applied for a cell composition.

Since what these procedures do is to generate some intermediate code to be interpreted later on, Pascal (or C) commands can be intermixed with calls to these procedures. The user can also define new procedures in terms of these basic procedures.

Fig. 9 shows the ALLENDE program, in C, that generates the PIF program of fig. 6. The generation of PIF code is straightforward: each call to one of the five procedures listed previously causes the generation of the corresponding code in PIF. For example, *endcell(" ")* generates only the character ")".



```
#include "/va/allende/usr/def.h"

main()
{
  begincell ( "example" );
    syscell ( CONTACT, nowire, nowire, diff(2), metal(3), Ø );
    place ( ABOVE );
    syscell ( CROSSING, poly(2), metal(3), poly(2), metal(3), Ø );
  endcell ( "   " );
}
```

Fig. 9 - An ALLENDE program

The procedure *syscell* allows the specification of a system cell. There are 10 kinds of system cells: CONTACT, ICONTACT, TRANSISTOR, ITRANSISTOR, PULLUP, CROSSING, LINE, JOG, EMPTY, and NULLCELL. These correspond to the system cells described in the previous section.

The only place where the user has to specify wires is for system cells, where he gives the wires at left, top, right, and bottom of the cell. The functions *metal(width)*, *poly(width)*, *diff(width)*, *nowire*, and the more general function *wire(layer,width)*, allow the specification of a wire (*metal(width)* is just a shorthand for *wire(METAL,width)*, for instance). Here is one place where a lot of parameterization can be done. *layer* and *width* can be parameterized; also, a wire can be a variable. It is also possible to have more than one wire at one side of the cell, allowing for overlapping wires or more complex cells.

The operators to be applied to the cells can be: LEFT, RIGHT, ABOVE, BELOW, ROTATED0, ROTATED90, ROTATED180, ROTATED270, FLIPPED0, FLIPPED90, FLIPPED45, FLIPPED135.

The procedure *extcell* specifies a filename containing an external cell. The external cell can be in intermediate form, in which case we call it *flexible*, or it can be in CIF, in which case we call it *rigid*. One special kind of external cells are pads, which are rigid cells. There is a pad library.

The procedures ***begincell*** and ***endcell*** are used to delimit a composite cell. The parameter of ***begincell*** is a character string containing the name of the cell; the name may be blanks only, in case we don't want to name the cell. The cell name is used to trace errors. If the cell is named it will correspond to a symbol in the CIF code, thus preserving the cell hierarchy, useful for programs that display CIF.

The parameter of ***endcell*** is a string containing the names of the parameter wires (usually only blanks). These names will appear in the CIF code, and they are useful for simulation.

Fig. 10 describes a ***nand*** cell, and gives some examples of parameterization. Fig. 11 describes a binary tree that uses the ***nand*** flexible cell generated previously.

```
program nand(output);
const
#include "/va/allende/usr/const.h"
type
#include "/va/allende/usr/type.h"

var power,ground,p2,d2: wiretype;

#include "/va/allende/usr/proc.h"

procedure contact(w1,w2,w3,w4: wiretype);
begin   syscell(CONTACT,w1,w2,w3,w4,0);   end;

procedure crossing(w1,w2: wiretype);
begin   syscell(CROSSING,w1,w2,w1,w2,0);   end;

procedure above;
begin   place(ABOVE);  end;

procedure nand;
begin
    begincell('nand');
        begincell('column1');
            syscell(LINE,power,nowire,power,nowire,0);  above;
            contact(nowire,nowire,p2,p2);               above;
            crossing(ground,p2);
        endcell('  ');
        place(LEFT);
        begincell('column2');
            contact(power,nowire,power,d2);             above;
            syscell(PULLUP,nowire,d2,p2,d2,4);          above;
            syscell(TRANSISTOR,nowire,d2,p2,d2,0);      above;
            syscell(TRANSISTOR,p2,d2,nowire,d2,0);      above;
            contact(ground,d2,ground,nowire);
        endcell('  ');
        place(LEFT);
        begincell('column3');
            crossing(power,p2);                         above;
            contact(p2,p2,nowire,nowire);               above;
            contact(p2,nowire,nowire,p2);               above;
            crossing(ground,p2);
        endcell('  ');
    endcell('  ');
end;

begin
    power:=  wire(METAL,5);
    ground:= wire(METAL,5);
    p2:=     wire(POLY,2);
    d2:=     wire(DIFF,2);
    nand;
end.
```

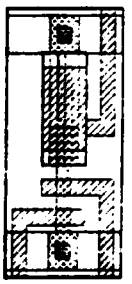


Fig. 10 - Nand cell

```
program binarytree(output);

const
#include "/va/allende/usr/const.h"
type
#include "/va/allende/usr/type.h"
#include "/va/allende/usr/proc.h"

procedure root;
begin
    extcell('nand.pif');
end;

procedure btree(n: integer);
begin
    begincell('btree');
        if n = 1 then root
        else begin
            root;
            place(ABOVE);
            begincell('  ');
                btree(n-1);
                place(LEFT);
                btree(n-1);
            endcell('  ');
        end (if);
    endcell('  ');
end;

begin
    btree(4);
end.
```

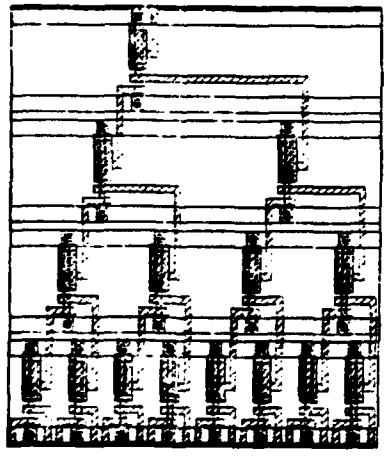


Fig. 11 - Binary tree

Parameterization can be done extensively, and it simplifies the modification of layouts. For example, a wire can be fully parameterized, as "power" in fig. 10. Parameters in the program not related to the ALLENDE basic procedures can also be used to produce general cells; one example is the depth of the tree in fig. 11.

Errors may occur during compilation of the user program, execution, interpretation of the PIF program, and solving the constraints. Compilation and execution errors are the usual ones, detected by the Pascal or C compiler or during execution. In case of error during interpretation of the intermediate form describing the layout, the PIF interpreter identifies the cell where the error occurred. The only possible error during the process of solving the constraints is a rigid cell not fitting the context where it is instantiated; in this case, the solver points to that cell which caused the error.

2.5 The Complete ALLENDE System

The ALLENDE layout system, as it stands now, is comprised of four programs:

- ALLENDE
  It takes a program and generates the layout (rigid cell), the circuit at the switch level, or a flexible cell to be used later on.

- SIMULATE
  This program is a switch-level simulator, based on [13].

- CIF2CELL
  The idea of CIF2CELL is to make possible the use of CIF code generated by other tools. It basically finds the cell interface. The CIF code is then used as a rigid cell.

- CIF2CIRCUIT
  When rigid cells are used, it is not possible to extract the circuit from the "high-level" description of the layout. In this case, the circuit is extracted from the layout in CIF. Our program interfaces to a circuit extractor, and currently we are using the Berkeley circuit extractor ***mextra*** [11].

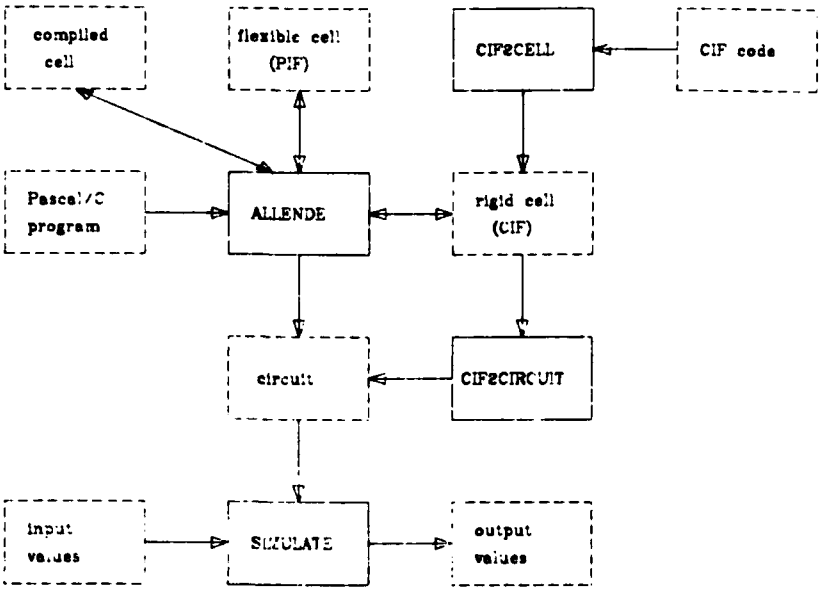


Fig. 12 - The ALLENDE system

The programs composing the ALLENDE system were written in C and Pascal. The system runs under the Berkeley UNIX operating system, and is currently used on a VAX 750.

3. Advantages of our approach

By making layout design similar to software design we can apply our knowledge about programming to this new activity. The main issues associated to the use of a procedural language for layout description are the following.

– **hierarchical design**

Hierarchy already exists in programming languages, in the form of procedures, and the programmer is used to it. Use of this hierarchy for layout design enforces a good design methodology.

- **expressive power**

  All the power of a procedural language is available to the designer. Parameters, conditional statements, iterative statements, etc., can be used. Parts of the layout, such as a PLA or a routing cell, instead of being described by the designer can be generated by a program.

- **parameterization**

  Having a layout design which produces different layouts for different values of a set of parameters is extremely useful. Examples are the parameterization of the layer or width of a wire, transistor ratios, or size of shift registers. This parameterization can involve local or global changes in the layout, and it simplifies the modification of layouts. It also allows general cells, whose characteristics depend on the values of some parameters (like a routing cell).

- **documentation**

  If the layout is described using a programming language we have some documentation on the design. This helps other people, and *even the own designer*, to understand the design.

- **open ended tool**

  Graphics editors tend to be closed tools in the sense that it is hard to automate the layout process beyond what the original design of the system allowed. Procedural languages are much better in this respect. The input to a compiler is text that can be generated by humans or by a program, while a graphics editor has an interactive nature, being designed basically to accept commands generated by humans.

- **no expensive equipment**

  With a programming language for layout description we can avoid the need for sophisticated computing resources. A standard alphanumerical terminal in combination with a small plotter or CRT display shared by several designers can be used effectively for layout design.

Much is gained by not assigning absolute positions to the layout elements directly, but by representing the relations among elements by a set of constraints. Implicit layout rules and cell flexibility are the main benefits of representing the layout as constraints. The design rules are implicit in the constraint generation process. This design rule free

environment relieves the designer of details that can cloud more global and important issues. What is more important, the layout obtained can be guaranteed to be free of design rule violations, thus eliminating the need for design rule checking.

If a piece of a layout is specified in absolute positions, serious problems are likely to arise when different pieces are put together. In constraint-based layout systems the absolute sizes or positions are determined by the system after solving the set of constraints. This makes cells flexible, with the possibility of being stretched in order to combine correctly with other cells.

4. Conclusions

The ALLENDE layout system has been used by a number of people in the design of chips which were successfully fabricated, and in experiments with layout tools [2]. Some layout tools, like a PLA generator and a pad router, have been written in ALLENDE with little effort.

One important aspect of a system, seen only when you use it, is detection and location of errors. In ALLENDE, the layouts produced are correct by definition, in terms of connectivity and design rules. In case of errors in the user specification of the layout, the system points the cells and wires involved in the error.

As far as compaction is concerned, the layouts produced by the system are relatively dense. It is hard to make a comparison of layout density for layouts produced by ALLENDE and those produced by hand, because that depends on the regularity of the layout and on the expertise of the designer. Based on our experiments, we find that for regular structures we obtain something close to the same density, while for irregular structures we spend about 20 percent more area than the corresponding hand-packed layout.

The structured representation of the layout and the use of an intermediate language (PIF) have led to a very efficient system, in terms of space and execution time, and a straightforward system implementation.

The ALLENDE layout system is based in the nMOS technology; this system was an experiment and the nMOS technology is well understood. There are plans to extend the language to the CMOS technology, and also to allow more layers, like a second metal layer. Besides that, we intend to investigate its applicability in the design of printed circuit boards.

A graphics editor can be easily incorporated to the ALLENDE system. The main characteristics of this editor, compared to other layout editors, would be:

- the objects dealt with by the designer are cells, and not shapes;
- the objects are composed in a structured way;
- the designer only specifies relative positions; the absolute positions are determined by the system, taking in account the design rules,
- the obtained layout is free of design rule violations; no checking is necessary;
- connectivity is also guaranteed, and the circuit can be directly obtained

In an ALLENDE program the circuit structure and the layout structure overlap, that is, the user describes at the same time both the circuit structure and the layout structure. The circuit structure goes down to the level of transistors and contacts. We could have a language allowing the specification of the circuit structure at a higher level (at the gate level or at the functional level, for example). From this specification the layout in PIF would be generated. Generalizing, PIF (or ALLENDE) could be the target language for a VLSI design tool, even a silicon compiler.

Besides being a powerful tool, ALLENDE is associated with a structured methodology for VLSI design. Having a tool that enforces hierarchy and the use of regular structures is going to improve the way we design integrated circuits. That is one step in the direction of managing the VLSI design complexity.

5. References

[1] R. W. Hon and C. H. Sequin
"A Guide to LSI Implementation - Second Edition"
Xerox PARC, Palo Alto, CA, Jan. 1980.

[2] K. Iwano and K. Steiglitz
"Some Experiments in VLSI Leaf-cell Optimization"
1984 IEEE Workshop on VLSI Signal Processing, Los Angeles, CA, Nov. 1984 (to appear).

[3] S. C. Johnson
"YACC: Yet Another Compiler-Compiler"
Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ, 1975.

[4] B. Kernighan and L. Cherry
"A System for Typesetting Mathematics"
Communications of the ACM, Vol. 18, No. 3, March 1975.

[5] T. Lengauer and K. Mehlhorn
"The HILL System: A Design Environment for the Hierarchical Specification, Compaction, and Simulation of Integrated Circuit Layouts"
1984 Conference on Advanced Research in VLSI, MIT, Jan. 1984.

[6] R. J. Lipton, J. Valdes, G. Vijayan, S. C. North, and R. Sedgewick
"VLSI Layout as Programming"
ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983.

[7] J. M. Mata
"Solving Systems of Linear Equalities and Inequalities Efficiently"
15th Southeastern Conference on Combinatorics, Graph Theory and Computing, Baton Rouge, LA, March 1984.

[8] J. M. Mata
"The ALLENDE Layout System User's Manual"
VLSI Memo #9, Princeton University, June 1984.

[9] J. M. Mata
"A Methodology for VLSI Design and a Constraint-Based Layout Language"
Ph.D. Thesis, Princeton University, Oct. 1984.

[10] R. Mathews, J. Newkirk, and P. Eichenberger
"A Target Language for Silicon Compilers"
COMPCON 82, San Francisco, CA, Feb 1982.

[11] R. N. Mayo, J. K. Ousterhout, and W. S. Scott
"1983 VLSI Tools: Selected Works by the Original Artists"
Report UCB/CSD 83/115, University of California at Berkeley, March 1983.

[12] J. Ousterhout
"CAESAR: An Interactive Editor for VLSI Layouts"
VLSI Design, Fourth Quarter 1981.

[13] V. Ramachandran
"An Improved Switch-Level Simulator for MOS Circuits"
20th Design Automation Conference, Miami Beach, FL, June 1983.

[14] S. Sastry and S. Klein
"PLATES: A Metric Free VLSI Layout Language"
1982 Conference on Advanced Research in VLSI, MIT, Jan. 1982.

[15] J. Valdes and R. L. Kalin
"ALI2 Documentation and Implementation Guide: Language Overview"
VLSI Memo #8, Princeton University, Feb. 1983.

[16] J. D. Williams
STICKS - A Graphical Compiler for High Level LSI Design"
AFIPS Conference Proceedings: 1978 National Computer Conference, Anaheim, CA, June 1978.

1984 IEEE Workshop on VLSI Signal Processing
University of Southern California
Nov. 12-14, 1984

Some Experiments in VLSI Leaf-cell Optimization†

Kazuo Iwano

Kenneth Steiglitz

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N. J. 08544

Abstract

This paper describes a method for local optimization of VLSI leaf cells, using the parameterized procedural layout language ALLENDE [5]. Tradeoffs among delay time, power consumption, and area are illustrated. Three different implementations of the 1-bit full adder are compared: a random logic circuit, a data selector, and a PLA. The fastest random logic 1-bit full adder has a time-power product about 1/3 that of the fastest data selector, and about 1/4 that of the fastest PLA. The 4-bit parallel adder is used to illustrate the effect of loading when leaf cells are combined.

1. Introduction

In the design of a custom VLSI chips it often happens that there is one cell that is used many times, usually in an array or a recursive structure. The fact that a cell is used many times means that there is a large potential payoff in its optimization, and that the problem can be made small enough to be manageable. Arrays of cells are especially common in digital signal processing applications, where regular structures, like systolic arrays, lead to designs that are easy to lay out efficiently, and have high throughput. As examples, bit-parallel and bit-serial multipliers can be constructed from one- and two-dimensional arrays of one-bit full adders, as can a wide variety of pipelined FIR and IIR filters (see [1], for example). As another example, a processor for updating one-dimensional cellular automata has been designed at Princeton which consists of a one-dimensional array of 5-input/1-output PLA's [10]. In such cases the problem of making most efficient use of a given piece of silicon breaks down into two distinct problems: 1) choice of the global packing strategy (the method of laying out and interconnecting leaf cells, and connecting them to power and clocks), and 2) the design of the iterated structure itself (which we call the *leaf cell*). In this paper we study the second problem: the design of efficient leaf cells. The example used throughout is the most common in digital signal processing, the 1-bit full adder.

There are three important measures of how good a leaf cell is: its time delay T; its peak or average power dissipation P_{max} or P_{ave}; and its area A.

† This work was supported by National Science Foundation Grants ECS-8307955, U.S. Army Army Research Office, Durham, NC, under Grant DAAG29-82-K-0095, DARPA Contract N00014-82-K-0549, and ONR Grant N00014-83-K-0275.

Ideally, the designer should be able to trade off these measures, one against the other. For example, in one application the clock may be fixed at a known value T_0, and it would therefore be senseless to make the the cell faster. On the other hand, peak power may be a real constraint because of heat dissipation limitations, and at the same time it may be important to keep the area small so as to fit as many cells on one chip as possible. We might therefore try to minimize some measure of the peak power and area (the product, for example), while enforcing the constraint $T \leq T_0$. In other applications speed may be critical, and it may be important to minimize T while observing constraints on P_p and A, and so on. In general, we would like to have enough information about the tradeoffs among the measures T, P and A to make intelligent design decisions. As we will see, the $P-T$ tradeoff is often of most interest, since the area is often a less sensitive function of design parameters (at least for fixed topology).

2. Formulation

The basic approach we take will be to search for local improvements on random initial designs. The search strategy will be to consider all single or double changes in element size along the critical path. When only single changes are tried, we call the procedure "1-change", when double changes are tried, "2-change". The idea is that the critical path indicates which parameters are most important to performance at any given point in the analysis.

We will limit the optimization to choice of pulldown widths. The method can be extended to choice of layers, orientation, and topologies. We will, however, study three radically different topologies for the full adder: the PLA, data-selector, and random logic.

The main analysis tools used in these experiments are the timing simulator CRYSTAL, and the power-estimation program POWEST, together with the rest of the Berkeley tool package [2].

Another essential component of the work is a procedural, constraint-based layout language for specifying VLSI layouts; in this case, we used the new language ALLENDE being developed at Princeton, a successor to ALI2 and CLAY [3,4,5]. This allows us to specify circuit parameters and have a cifplot generated automatically.

3. The Critical-Path Optimization

Figure 1 shows how the optimization is performed in our experiments. In Figure 1 **faparm** is an input parameter vector to **ANALYSIS** which has diffusion widths of nodes as described in section 4. The initial **faparm** is generated at random by **RANDOM** according to its input file **pattern**. **ANALYSIS** takes **faparm** as its input and generates an appropriate layout and its resulting **T**, **P**, and **A**, as well as **the nodes on the critical path** (hereafter called the critical path nodes). Since every node on the critical path has an associated parameter in **faparm**, **CASEGEN** can generate **faparms** as subcases by using the one-(two-)change method. Here the one-(two-)change method changes one(two) parameter(s) associated with the critical path nodes by one step. (From here on the 1-change method is denoted by 1-opt or Random 1-opt, and the 2-change method by 2-opt or Random 2-opt.)

The optimization strategy is shown in the flowchart of figure 2. When the first improvement occurs, this case is picked up for the next iteration. If no improvement occurs but there exists a case which has the same cost and has

not yet been analyzed, this case is adopted next. Otherwise a new random **faparm** is generated for the next iteration, to search for other locally optimal points. We used two cost criteria for optimization: T, and $P_{max}T$ (hereafter denoted by PT). Figure 3 shows an outline of the main procedures used in the **ANALYSIS** loop. A short description of each follows below:

1) **ALLENDE** This procedural constraint-based VLSI layout language produces an integrated circuit layout in Caltech International Form (CIF) corresponding to the specified parameters [5].

2) **MEXTRA** MEXTRA reads CIF and extracts the nodes to create a circuit description for further analyses [2].

3) **CRYSTAL** CRYSTAL is used for finding the worst-case delay time of the circuit [2].

4) **POWEST** POWEST is used for finding the average and maximum power consumption of the circuit.

5) **CRITICAL** CRITICAL reports the critical path nodes by using the output of CRYSTAL.

6) **LIST** This command stores the vector of results (**T,P,A**) in the HISTORY file for further optimization.

In figure 3 the squares surrounded by dotted lines are files used for inputs or outputs of the above procedures.

1) **faparm** The faparm has parameters for layout generation; for example, the diffusion width of each node, the permutation of product terms in a PLA, etc.

2) **layout generating program** There are several ALLENDE programs implementing desired circuit topologies such as the PLA, random logic, etc. Each program requires parameters in its corresponding faparm.

3) **the critical path nodes** The critical path nodes are extracted from the output of CRYSTAL. Each node can be associated with parameters in faparm. This is done by looking up a table for each topology, which associates each node with its corresponding parameter.

4. Full-Adder Circuit Implementations

As mentioned in the Introduction, we adopted the 1-bit full-adder circuit as an example for experimentation, because it is relatively simple, but is a basic arithmetic logic circuit. The 1-bit full-adder circuit can be implemented in many ways. We chose three kinds of circuits: the **PLA**, **Data Selector**, and **Random logic**. Each layout has several parameters. We will use the vector representation of these parameters; that is $d = (d_1, d_2, \ldots, d_n)$ means that the *diffusion width of node i is* $d_i\lambda$. *We also use the vector* $k = (k_1, k_2, \ldots, k_n)$ to mean that the pullup to pulldown ratio of the inverter, NOR, or NAND circuit in which node i exists is k_i. The vector k is fixed for each circuit.

1) **PLA**

Figure 4 shows the full-adder circuit diagram implemented by a programmable logic array (PLA) [7]. This layout has the following 17 parameters and 2 permutations.

$d = (d_{and_1}, \ldots, d_{and_7}, d_{or_1}, d_{or_2}, d_{in_{1,1}}, \ldots, d_{in_{3,2}}, d_{out_1}, d_{out_2}, \pi_1, \pi_2)$

$k = (4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4)$

- 7 pulldown diffusion widths of the AND plane.
- 2 pulldown diffusion widths of the OR plane.
- 6 pulldown diffusion widths for inputs.
- 2 pulldown diffusion widths for outputs.
- 1 permutation of product terms in the AND plane.
- 1 permutation of outputs.

In the optimization process, the two permutations are fixed for the sake of simplicity. However those two permutations are chosen in advance in order to give the best result before the optimization by doing experiments based on various random permutations as inputs.

2) **Berkeley PLA**

The PLA generated by using **mkpla** of the Berkeley VLSI tools [2,8] is used for the purpose of cost comparison with the PLA implemented in 1). This PLA is not optimized, but uses the following fixed parameter vector.

$d = (4,4,4,4,4,4,4,4,4,4,8,8,8,8,8,8,8,8)$

$k = (4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4)$

3) **Data Selector**

Figure 5 shows the full-adder circuit diagram of a Data Selector implementation [9]. The following truth table is used.

C_i	B	S	C_o
0	0	A	C_i (or B)
0	1	$\bar{A}$	A
1	0	$\bar{A}$	A
1	1	A	C_i (or B)

This circuit selects inputs (A, $\bar{A}$, or C_i) instead of calculating S and C_o. Here C_i is the input carry signal, C_o is the output carry signal, and S is the output sum signal. A and B denote the two other inputs. This layout has the following 8 parameters.

$d = (d_A, d_B, d_{C_i}, d_1, d_2, d_3, d_{C_o}, d_S)$

$k = (4,4,4,4,8,4,8,8)$

- 3 pulldown diffusion widths for input inverters.
- 3 pulldown diffusion widths for internal inverters.
- 2 pulldown diffusion widths for output inverters.

4) **Random Logic**

Figure 6 shows the circuit diagram of the Random Logic Implementation [6].

This layout has the following 4 parameters.

$d = (d_1, d_2, d_{C_0}, d_S)$

$k = (8, 12, 4, 4)$

- 2 pulldown diffusion widths for internal inverters.
- 2 pulldown diffusion widths for output inverters.

All the circuits above were verified by **ESIM** [2] or **SIMULATE** [5].

5. Parameterization

The diffusion width of the pullup in each stage is automatically determined and implemented by ALLENDE in the following way. Suppose that the current parameter vector is $d = (d_1, d_2, \ldots, d_n)$, and the pullup-to-pulldown ratio vector of the specified layout is $k = (k_1, k_2, \ldots, k_n)$. (The choice of pullup-to-pulldown ratio is discussed in [7].) For each node i, define the variables Z_{pu}, Z_{pd}, and a pullup-to-pulldown ratio K as follows.

$$Z_{pu} = \frac{L_{pu}}{W_{pu}}, \quad Z_{pd} = \frac{L_{pd}}{W_{pd}}, \quad K = \frac{Z_{pu}}{Z_{pd}}$$

where
L_{pu} (L_{pd}) is the length of pullup (pulldown).
W_{pu} (W_{pd}) is the width of pullup (pulldown).
$W_{pd} = d_i$, $K = k_i$ and $L_{pd} = 2$.

L_{pu} and W_{pu} are determined as follows.

If $W_{pd} \leq 2K$

$$W_{pu} = 2$$
$$K = \frac{L_{pu} / 2}{2 / W_{pd}} \quad \text{or} \quad L_{pu} = \frac{4K}{W_{pd}}.$$

If $W_{pd} > 2K$

$$W_{pu} = W_{pd} / K$$
$$K = \frac{L_{pu} / W_{pu}}{2 / W_{pd}} \quad \text{or} \quad L_{pu} = \frac{2KW_{pu}}{W_{pd}}.$$

We adopted following choices.

1) $\lambda = 2\mu$
2) The timing estimation program CRYSTAL uses an input pulse which is 1 *nsec* wide.

6. Results

Table 1 shows a comparison of the performance of our implementations. Each row represents one locally optimal point using as criterion the item indicated by *. The units of A, P_{ave}, P_{max}, T, APT and PT are λ^2, $(10^{-6} * W)$, $(10^{-6} * W)$, ns, $(10^{-12} * \lambda^2 * W * ns)$ and $(10^{-6} * W * ns)$ respectively in all tables. Figure 7 shows P_{max} vs T curves for different topologies, while figure 8 shows several P_{max} vs T trajectories obtained during the process of optimization using the 1-change and 2-change methods for the Data Selector and the Random

Table 1. performance comparison (1 bit full adder)

type	A	P_{ave}	P_{max}	T	APT	PT	parameter
PLA	21560	6472	10183	12.8*	2802	1303	1)
	21840	5678	9241	15.3*	3087	1413	2)
	21762	5503	8616	14.9*	2794	1284	3)
PLA(Berkeley)	22176	7314	11749	12.8*	3339	1504	4)
Data Selector	8100	3765	6117	15.8*	783	966	8 8 8 8 8 8 8 8
	8100	3529	5645	16.5*	754	931	8 8 8 4 8 8 8 8
	8190	3764	6116	15.9*	796	972	12 8 8 8 8 8 8 8
Random Logic	7742	1331	1957	16.5*	392	323	16 12 3 2
	9600	1683	2427	16.4*	382	398	16 24 2 3
	9800	1644	2329	16.4*	378	382	16 24 2 2
	9600	1723	2506	16.5*	397	413	16 24 3 3
	5194	705	1096	22.6	128	248*	6 8 2 2
	4704	626	1018	25.9	124	264*	4 6 3 2
	5136	744	1174	22.9	138	269*	6 8 2 3

1) d = (4,4,4,4,4,4,3,4,4,8,8,8,4,4,4,8,2)

2) d = (4,2,3,3,3,3,3,4,3,8,8,8,4,4,4,8,2)

3) d = (3,3,3,4,4,4,4,3,3,8,8,8,4,4,4,4,3)

4) d = (4,4,4,4,4,4,4,4,4,8,8,8,8,8,8,8,8)

Table 2 performance comparison (4 bit parallel adder)

type	A	P_{ave}	P_{max}	T	APT	PT	parameter
Data Selector	41310	16536	28218	75.3*	877761	212482	4 8 8 8 16 8 16 16
	44550	16536	28218	84.1*	1057230	237313	4 8 8 8 16 8 24 16
	45409	16534	28213	84.3*	1079990	287836	4 8 8 16 16 16 16 16
	44523	13248	21641	91.0*	876805	196933	4 8 8 8 16 4 8 4
	42845	12301	19748	92.5*	782645	182669	4 8 4 4 16 8 8 4
	43747	11362	17868	94.9*	741806	169567	4 8 4 4 16 4 8 4
	43605	12354	20692	98.0*	884229	202782	2 8 4 8 16 8 4 8
	45441	11885	19753	100.8*	904777	199110	2 8 4 8 16 8 4 4
	44523	12305	19755	101.1*	889227	199723	4 8 4 8 16 4 4 8
	44649	11831	18808	103.2*	866631	194099	4 8 4 4 16 8 4 4
	43747	11362	17868	103.6*	809812	185112	4 8 4 4 16 4 4 8
Random Logic	35552	6577	10335	41.1*	151014	42476	16 12 8 2
	34848	6734	10649	41.4*	153634	44087	16 12 8 3

Logic circuit. Each point takes about 1.5 minutes of cpu time on a VAX 11/750. Many of the locally optimal solutions have identical parameter values on the critical path, but differ in other coordinates because of different random starting values.

7. Parallel Adder : The effect of loading factors

The preceding results did not take the loading on the output of the circuit into account. When these circuits are used in arrays, this may become important. To study this problem, we implemented two circuits for a 4-bit parallel adder, using the **Data Selector** and the **Random Logic** 1-bit full adders of the previous section. The results are shown in Table 2.

8. Discussion of Results

8.1. P_{max} vs T tradeoff

Figure 8 shows P_{max}-T trajectories followed by the critical path optimization process, when minimizing T for the **Random Logic** circuit. The dotted envelope shows the final tradeoff curve for P vs T. Notice that the locally optimal point obtained by using PT as the cost criterion lies very close to the trajectory obtained when minimizing T. (See point **a**, with $P = 12.5mW$, and $T = 22.4ns$.) For comparison, the optimization for PT gave us a locally optimal point **b** with $P = 10.9mW$ and $T = 22.6ns$, very close to point **a**. Thus, optimization using the two criteria is consistent.

8.2. Performance comparison among the PLA, Data selector, and Random logic.

Table 3 normalized performance comparison (1-bit full adder)

type	A	P_{ave}	P_{max}	T	APT	PT
Random Logic	100	100	100	100	100	100
Data Selector	105	283	313	96	200	299
PLA	278	486	520	78	715	403
PLA(Berkeley)	286	550	600	78	852	466

Table 3 shows a normalized performance comparison of the best locally optimal point for each layout, minimizing T. The **Random Logic** seems to be the best choice in all respects except T. However, it is the fastest among the 4-bit parallel adder implementations. The T of the 4-bit parallel adder using **Random Logic** is less than 4 times the T of the 1-bit full adder, while in the other layouts it is more than 4 times the T of the 1-bit full adder. The reason is that this **Random Logic** 1-bit full adder circuit calculates the carry signal and propagates it before the calculation of the sum signal, so the carry ripple propagates faster than the sum. As a result, the 4-bit parallel adder takes only 2.5 times as much time as the 1-bit full adder. Figure 7 shows the P-T tradeoff curve of each layout. The curve for the **Random Logic** circuit is below the one for the **Data Selector**, which is below that for the **PLA**. Hence we can order the layouts with **Random logic** best, **Data Selector** next, and **PLA** last. This result agrees with our intuition because this order is the same as the order of circuit specialization.

8.3. Comparison between our PLA and the Berkeley PLA

Both **PLA**'s have almost the same costs, except for P. The reason is that our locally optimal point occurs at the choice d = (4,4,4,4,4,4,3,4,4,8,8,8,4,4,4,8,2),

while the **Berkeley PLA** adopts d = (4,4,4,4,4,4,4,4,4,8,8,8,8,8,8,8,8). The **Berkeley PLA** is therefore very close to locally optimum with respect to T.

8.4 Comparison with Myers' work

Myers did similar performance comparisons of various 1-bit full adder implementations [9], but did not use any optimization. His results, shown in Table 4 below, are quite different from ours, shown in Table 3. Our results show that an appropriate choice of layout and its optimization makes the **Random Logic** circuit better than the **Data Selector**, and that the **PLA** can be made very fast at the expense of Power.

Table 4. 1-bit full adder normalized performance comparison (Myers[9])

type	A	P_{max}	T	APT	PT
Random Logic	100	100	100	100	100
Data Selector	45	50	125	28	72.5
PLA	105	110	170	196	187

8.5. 4-bit Parallel Adder

Tables 1 and 2 show that the locally optimal point of the 1-bit full adder is attained with a pull-down diffusion width of the carry output stage d_{C_0} = 2 or 3, while the corresponding width for the 4-bit parallel adder is d_{C_0} = 8. The pullup width remains 2. This suggests that the critical path passes through the pull-down of the output carry stage, which is indeed the case.

On the hand, for the **Data Selector**, the critical path passes through the pullup of the output carry stage, and in fact it is the pullup width that expands during optimization of the 4-bit parallel adder.

8.6. Comparison of the 1-change and 2-change methods

Figure 8 and Table 5 show a comparison between the 1-change and the 2-change methods when applied to the **Random Logic** implementation. Table 5 is discussed in the next section. The slope of the 2-change method is steeper than that of the 1-change method, but the 2-change method reaches better locally optimal points. Hence in this case the 2-change method works better than the 1-change method does. However, the 2-change method does not work as well as the 1-change method for the **Data Selector**, which has many more parameters. The 2-change method took more iterations than the 1-change method and did not obtain better locally optimal points.

8.7. Effectiveness of our optimization: Cost Improvement ratio

Table 5 below shows the average initial delay times T_0 (obtained from random starts), the average locally optimal delay time T_{opt}, the average percent improvement of the delay time T, and the best locally optimal delay time T_{best}. We can see from this that **2-opt** performs much better than **1-opt**. We should note that it is very important to choose a good order in which to try improvements, because this saves unnecessary search time evaluating changes that are unlikely to be improvements. For example, we chose the diffusion widths of the 3-input NAND gate as the first parameters tried for the **Random Logic** circuit.

Table 5 Cost improvement of our optimization methods

type	opt	criterion	$T_{initial}$	T_{opt}	% improvement	T_{best}
Random Logic	1-opt	T	29.7	19.2	33	19.1
Random Logic	2-opt	T	29.7	16.8	42	16.4
Data Selector	1-opt	T	24.3	17.7	25	15.8
Data Selector	2-opt	T	23.5	18.0	23	15.8
PLA	1-opt	T	19.3	16.3	16	12.8

9. References

[1] P. R. Cappello, K. Steiglitz, "Completely Pipelined Architectures for Digital Signal Processing," *IEEE Trans. on Acoustics, Speech, and Signal Proc.*, vol. ASSP-31, No.4, pp. 1016-22, Aug. 1983.

[2] R. N. Mayo, J. K. Ousterhout, W. S. Scott, "1983 VLSI Tools," Report No. UCB/CSD 83/115, Computer Science Division (EECS), University of California, Berkeley, Calif., March 1983.

[3] S. C. North, "Molding Clay: A Manual for the CLAY Layout Language," VLSI Memo #3, EECS Department, Princeton University, Princeton, N. J., July 1983.

[4] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, G. Vijayan, "VLSI Layout as Programming," *ACM Trans. on Programming Languages and Systems*, July 1983.

[5] J. Mata, "ALLENDE User Manual," VLSI Memo #9, EECS Department, Princeton University, Princeton, N. J., May 1984.

[6] R. Rondell, P. C. Treleaven, *VLSI architecture*, Prentice-Hall Inc., Englewood Cliffs, N. J., 1983.

[7] C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Co. Menlo Park, Ca., 1980.

[8] J. Mata, "A PLA Generator for the ALLENDE Layout System," EECS Department, Princeton University, Princeton, N. J., June 1984.

[9] D. J. Myers, "Multipliers for LSI and VLSI Signal Processing Applications," Masters Degree thesis, Edinburgh University, Edinburgh, England, Sept. 1981.

[10] R. R. Morita, "Pipelined Architecture for a Cellular Automaton," Senior Independent Project Report, EECS Department, Princeton University, May 1984.

Figure 1. Overall flowchart.

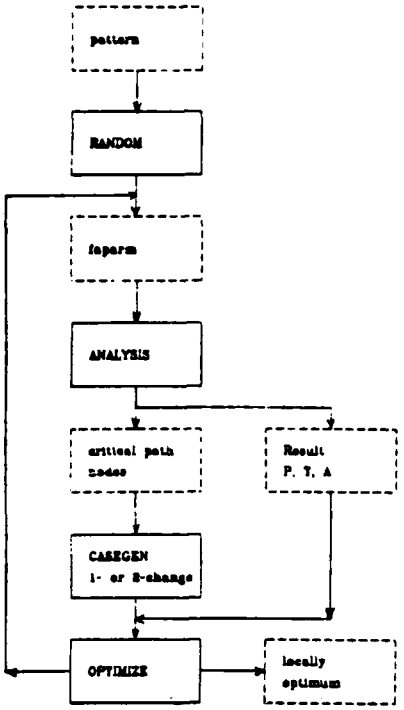


Figure 2. Flowchart of the critical path optimization method.

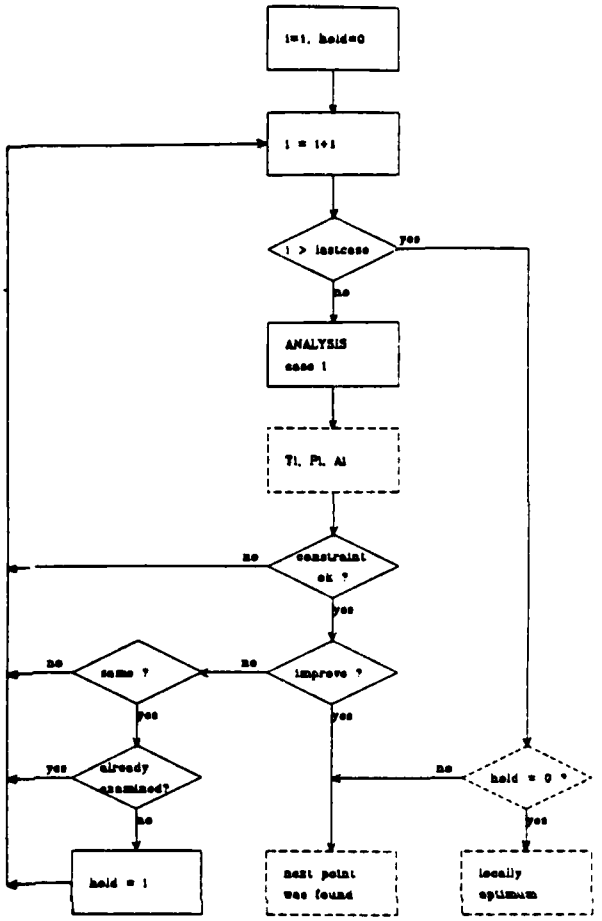
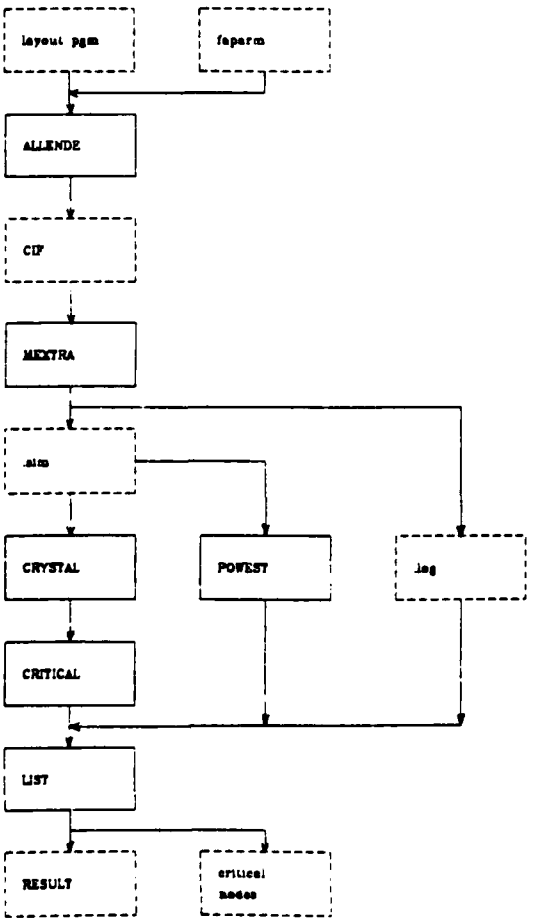


Figure 3. Detail of the ANALYSIS procedure.



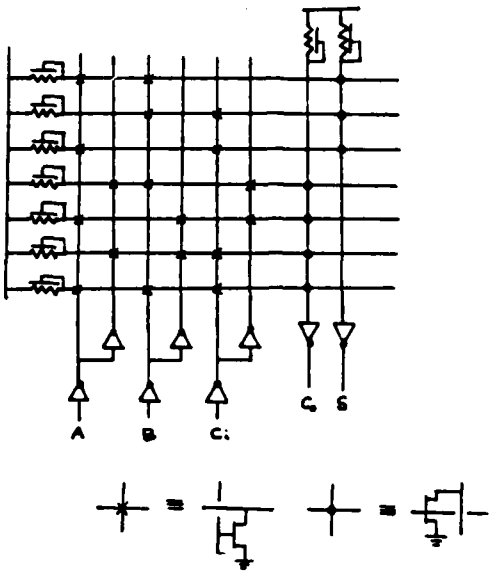


Figure 4. PLA

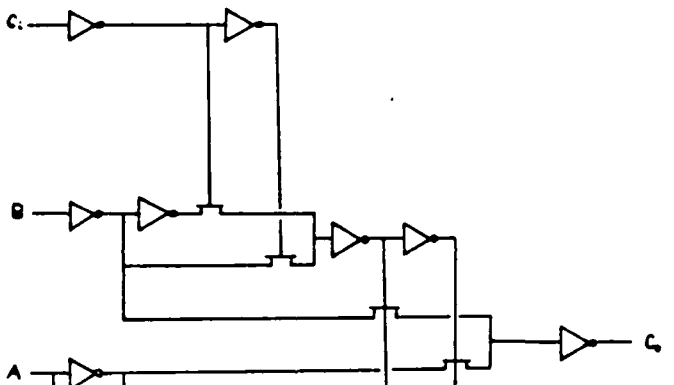


Figure 5. Data Selector

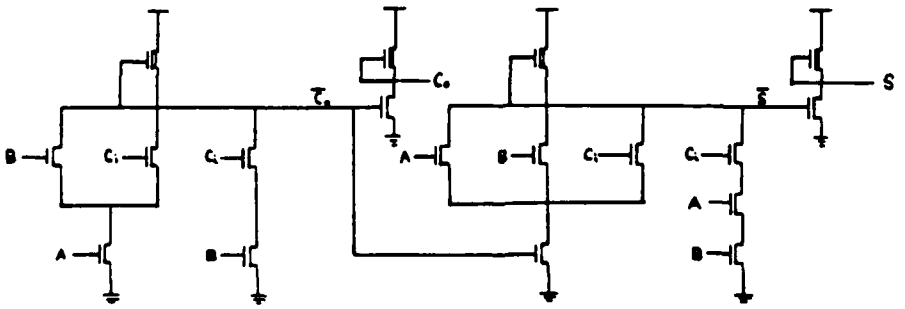


Figure 6. Random Logic

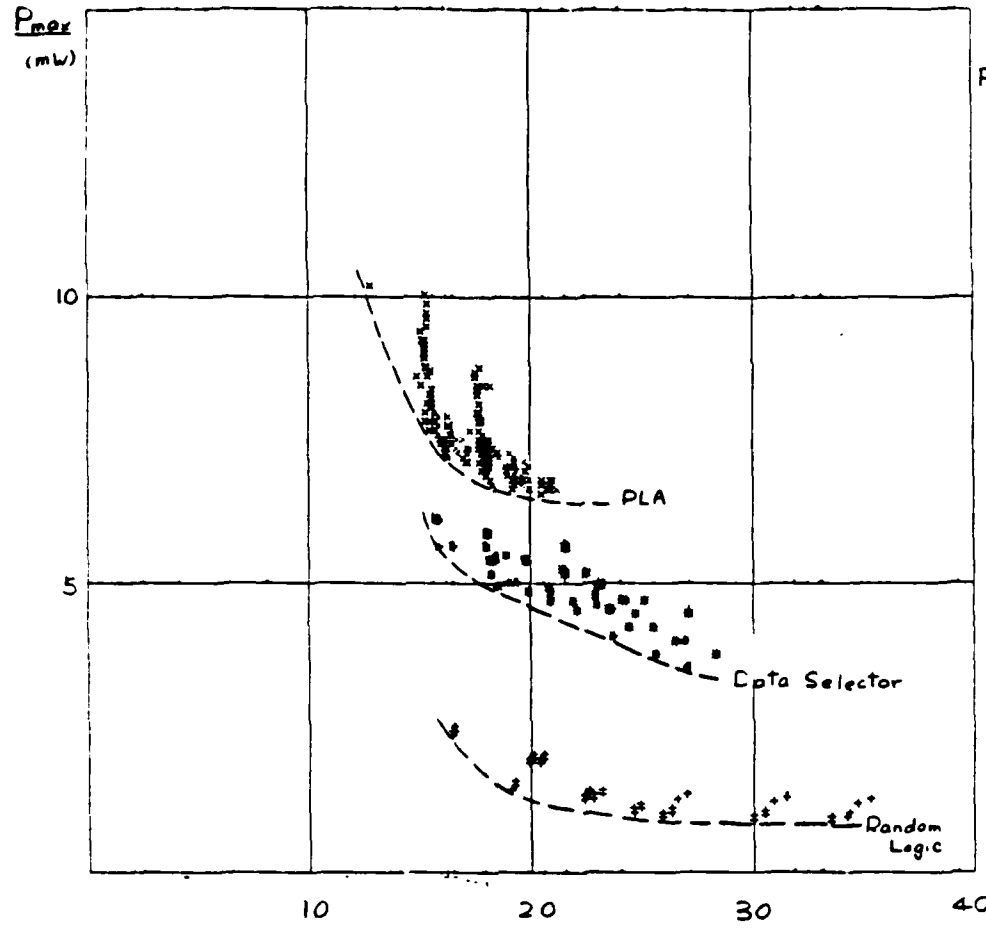


Figure 7

× PLA

* Data Selector

+ Random Logic

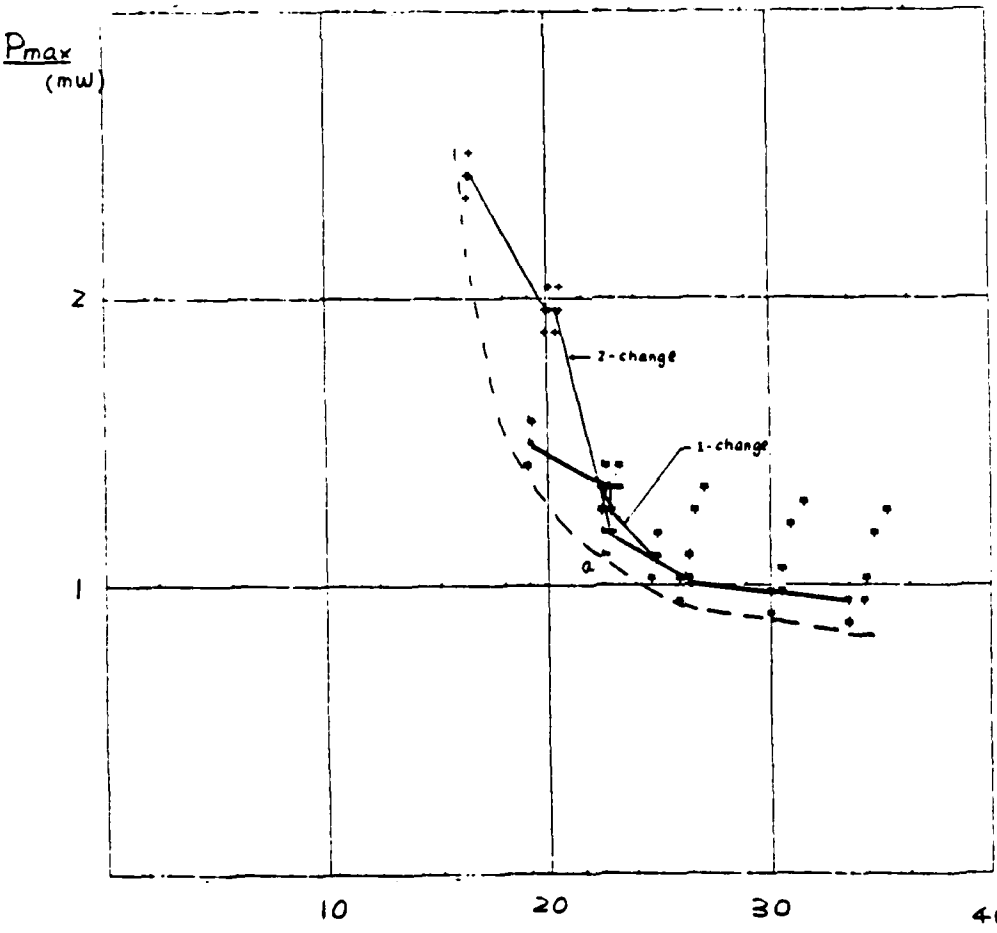


Figure 8

+—+ 2-change

=—= 1-change

Fifth IMACS International Symposium On
COMPUTER METHODS FOR
PARTIAL DIFFERENTIAL EQUATIONS
19, 20, 21 June 1984 Lehigh University - Bethlehem, Pennsylvania 18015 - USA

Efficiency of Parallel Processing in the Solution of Laplace's Equation†

William C. Moore

Information Systems Laboratory
Dept. of Electrical Engineering
Stanford University
Stanford, California 94305

Kenneth Steiglitz

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

ABSTRACT

A parallel processing architecture for the solution of partial differential equations by point iteration is studied. Grid points are stored in a circulating memory and identical processors are spaced around the store. Computer simulation of the solution of Laplace's equation with a simple point iteration relaxation algorithm for one-, two-, and three-dimensional problems shows that convergence rates intermediate between those of the Jacobi and Gauss-Seidel methods are obtained. Hardware utilization efficiency (speedup relative to the number of processors) of 40-60% is achieved with as many as N processors, where N is the number of non-boundary grid points. Furthermore, for up to $N/2$ processors, the efficiency remains above 90% in the one-dimensional case, and above 75% in the two-dimensional case. There are sharply diminishing returns for using more than $N/2$ processors.

1. Introduction

The solution of partial differential equations taxes the largest and fastest present-day general-purpose computers. Physically meaningful problems often need huge amounts of time and space. Clearly, with the decreasing cost of large-scale integrated circuits, it seems profitable to build special-purpose devices for solving partial differential equations which use many identical processors operating in parallel. This paper describes a study of a circular arrangement of processors and a circulating store. Simulation results for the very simplest numerical problem are described: the solution of Laplace's equation with Dirichlet boundary conditions, using point iteration methods.

When explicit, point iterative methods are used to solve partial differential equations, a grid point value is updated by replacing it with some function of the values at neighboring points in the grid. In the system described here, the grid of points is mapped by a raster scan into a circulating serial bit stream. The bit stream passes through processors that update the grid points as they pass through the processors. Using this approach, problems with arbitrary numbers of dimensions can be treated. In addition, identical processors can be added without reorganization.

In the Gauss-Seidel method [8] the grid point values are updated in an orderly, row-by-row and plane-by-plane fashion, and new values are used as soon as they become available. In the Jacobi method [8] old grid point values are used throughout each iteration. When the circulating store system uses one processor, the calculation reduces to the Gauss-Seidel method; when it uses one processor per grid point, it reduces to the Jacobi method. When a number of processors between these two extremes is used, there is a complicated mixture of old and new values used by the processors. The purpose of this paper is to investigate experimentally the rate of convergence as a function of the number of processors, and thereby to evaluate the potential hardware utilization efficiency of the circulating store system. The results show that the speed of convergence is, as might be expected, intermediate between the Gauss-Seidel and Jacobi methods.

2. Circulating Store Configuration

We study here a synchronous circuit consisting of a long shift register arranged in a circle (the main memory of the system), and a number of independent processors tapping and updating the stream at various points (see Fig. 1). Similar configurations have been suggested by various workers at different times for applications such as Monte Carlo calculations [1] and image processing [2], as well as partial differential equations [3,4,5]. A fixed network of microprocessors which communicate locally has also been studied [6,7], and these two arrangements are equivalent when there is one processor per grid point.

If each grid point is mapped to a set of contiguous bits in the stream, some bits in the set can represent the value of the function at the grid point, while the remaining bits can be used to flag boundary values,

†This work was supported by NSF Grant ECS-8120037, U. S. Army Research-Durham Grant DAAG29-82-K-0095, DARPA Contract N00014-82-K-0549, and ONR Grant N00014-83-K-0275.

store space-dependent coefficients, and possibly hold other information. We will call the set of bits in the bit stream corresponding to a grid point simply a "grid point value." At any given time each processor must be able to change the value of the grid point which it will update, as well as read the values from those grid points whose values are needed to do the update calculation. For example, with a 5-point molecule in the Gauss-Seidel method for a two-dimensional Laplace equation, each processor must have information from 4 neighboring points, as shown in Fig. 2.

There is no direct communication between processors. With each *major clock cycle*, the circular shift register shifts one complete grid point, each processor reads the necessary information, and if the grid point currently associated with a processor is not a boundary value, it is updated. Each grid point can have a bit reserved to indicate convergence, based on the change from the previous value of the function, and that bit can be kept current every time the value is updated. A counter can then be inserted in the circulating store to detect the condition where all the points have converged. Alternatively, a global counter can receive this information from every processor every major clock cycle. We will assume this latter method in the simulation because it detects convergence sooner and gives finer resolution in the measurement of running time, but obviously it is not necessary for the operation of the device.

Note that no additional time is required to observe the changing grid point values on a graphics screen, since the bit stream can be passed serially through such a display device without interfering with the calculation.

We emphasize that the processors work in parallel, and so the answer from one processor is not available until the next major clock cycle. Thus, as was pointed out before, the values used in any one calculation are in general of various ages.

3. Method of Performance Evaluation

When all grid points have passed through all processors once, we say that one *iteration* has taken place. This corresponds to each bit in the stream being shifted all the the way around the circle. The time required for this 360° shift depends on the major clock cycle time and the number of grid points. (We assume that a processor completes its function during one major clock cycle.) Since the processors operate in parallel, the time does *not* depend on the number of processors; an iteration represents an amount of time that is independent of the number of processors. Neglecting such things as time for loading boundary values, the number of iterations required for convergence is a reasonable measure of real time required for convergence, and can be used to compare the performance of different systems. However, for systems with different grid sizes or representing different equations, an iteration may mean different things and thus cannot serve as a basis for comparison. Note also that there is no reason that the number of iterations required for convergence need be an integer. (Recall that we are using a global counter to detect convergence.)

Since the number of iterations required for convergence is proportional to the time required for convergence, a reasonable idea of the performance of a system as a function of number of processors can be obtained by investigating the relationship between the number of iterations required and the number of processors. If we let $its(n)$ be the number of iterations required with n processors, we can define

$$e(n) = \frac{its(1)/n}{its(n)}$$

to be the *efficiency* with n processors, the efficiency with one processor being 100%. In general the efficiency will be less than 100%, but it is not impossible for it to exceed 100% (e.g. two processors can be more than twice as fast as one).

4. Test Problem: Laplace's Equation

A computer simulation of the scheme described above has been carried out using Laplace's equation ($\nabla^2 f = 0$) on a line, in a square, and in a cube, with Dirichlet boundary conditions. Explicit iterative methods for Laplace's equation are widely used, and their convergence characteristics are well known. (See [8], for example.) The one used in the simulation is the simplest: Each grid point is replaced by the average of the points immediately adjacent (not diagonal) to it. Thus, for a k-dimensional problem, a point is replaced by the average of the $2k$ points adjacent to it on the rectilinear lattice of grid points in k-space.

As suggested above, the grid is mapped to a serial stream by using a raster scan; the end of one line is connected to the beginning of the next. Some experiments indicated that using other scanning patterns, such as boustrophedon (back and forth, as the ox plows) has little if any effect on the results.

The convergence criterion used is based on the maximum relative change in function value at the grid points. If the old and new values at grid point k are respectively $g^t(k)$ and $g^{(t+1)}(k)$, then we say we have converged at point k if at the most recent update at point k we have

$$|g^{(t+1)}(k) - g^t(k)| < \varepsilon\, g^t(k)$$

where ε is the convergence criterion. If at some moment we have converged at all grid points, we say the computation itself has converged.

Problems with a variety of different dimensions, grid sizes, boundary values and tolerances were simulated and we next present some numerical results.

5. Experimental Results

Figures 3-5 show plots of efficiency $e(n)$ vs. n for three typical problems, of one-, two-, and three-dimensions. In all three cases the convergence tolerance is $\varepsilon = 0.002$, and non-boundary grid points have the initial value 0.

The one-dimensional grid has 200 points, including boundary points. One boundary value is 0.0 and the other 1.0. The two-dimensional grid is 20×20 points, with the square boundary having the value 1.0. The 10×10×10 point three-dimensional case also has its boundary values equal to 1.0 everywhere.

As expected because of the gradual transition between the extreme cases of the Gauss-Seidel and Jacobi methods, there is a general downward trend in

efficiency. Furthermore, the efficiency decreases from 100% to about 50%, as would be expected from the fact that the Jacobi method for this problem is theoretically asymptotically one-half as fast as the Gauss-Seidel [8]. An efficiency of 50% with n processors means that we are converging $n/2$ instead of n times as fast as with one processor.

The different dimensions give rise to different curve shapes, but those shapes did not vary much as convergence tolerance, grid size, and boundary conditions were varied. In the one-dimensional case, efficiency is near 100% as long as the number of processors is less than half the number of non-boundary grid points, but at that point, efficiency falls off sharply to about 50% with 200 processors. In two dimensions the efficiency curve has two fairly distinct levels, with the break point again coming at approximately half the number of non-boundary grid points. The efficiency plot for three dimensions seems not to have two distinct regions, but falls off gradually. In all cases there is a great deal of local jumping up and down, due evidently to the particular way in which the processors use the information of neighboring processors in particular arrangements.

The maximum absolute speed is obtained by having N processors, where N is the number of non-boundary grid points, but there are diminishing returns for using more than about $N/2$ processors. In any particular application the choice of number of processors will depend on the cost of a single processor relative to the cost of the whole system. The efficiency with N processors remains above 40%, sometimes even getting as high as 65%.

6. Over-relaxation

A preliminary test was made of a simple over-relaxation strategy in the one-dimensional case. Here the new value at each grid point is defined by

$$g^{(t+1)} = g^t + \alpha(g_c^{(t+1)} - g^t)$$

where $g_c^{(t+1)}$ is the value that would be adopted at this step if over-relaxation were not being used, and $\alpha \geq 1$ is the over-relaxation parameter. When α is taken to be 1.5 in the one-dimensional case described in Fig. 3, the number of iterations required by one processor is reduced from 397,584 to 212,057, an increase in absolute speed of 87%. Figure 6 shows a plot of the efficiency vs. n for $\alpha = 1.5$. It is of the same general character up to $N/2$, showing efficiency near 100% in this range, but past that point the iteration rapidly becomes unstable (with the efficiency therefore going to zero). Further work is needed to explain and predict the stability of the over-relaxation method for the parallel computation scheme discussed here.

7. Conclusions

The simulation results for the circulating store method and the standard point iteration method are in accord with theory, and they are encouraging: n processors never operate slower than about $n/2$ times as fast as one. Furthermore, for up to $N/2$ processors, where N is the number of non-boundary grid points, the efficiency remains above 90% in the one-dimensional case, and above 75% in the two-dimensional case. There are sharply diminishing returns for using more than $N/2$ processors.

The approach is applicable to linear and nonlinear problems of any dimension with any boundary conditions, makes efficient use of large numbers of identical processors, and has a very simple, linear, interconnection pattern. More work is needed to determine the stability and convergence rates of the over-relaxation method, and more sophisticated and potentially faster methods, in higher dimensions, for more ambitious problems.

8. Acknowledgments

The circulating store configuration for solving differential equations was developed with the co-authors of [5]: J. Bruno, A. C. Davis, M. Kostin, and C. Wyman. Also, we thank R. J. Lipton for helpful comments.

9. References

[1] R. B. Pearson, J. L. Richardson, and D. Toussaint, "A Special Purpose Machine for Monte-Carlo Simulation," Institute for Theoretical Physics Report NSF-ITP-82-98, University of California, Santa Barbara, California, 1981.

[2] C. Rieger, "ZMOB: A Mob of 256 Cooperative Z80A-Based Microcomputers," Conference paper, Computer Science Department, University of Maryland, College Park, MD 20742.

[3] C. T. Leondes, and M. Rubinoff, "DINA, a Digital Analyzer for Laplace, Diffusion and Wave Equations," *Trans. AIEE*, Pt. 1, Vol. 71, Nov. 1952, pp. 303-309.

[4] R. F. Rosin, "A Special Purpose Computer for Solution of Partial Differential Equations and other Iterative Algorithms," *IEEE Trans on Electronic Computers*, June 1965, pp. 488-490.

[5] Bruno, J., Davis, A. C., Kostin, M., Steiglitz K. and Wyman, C., "Linear Organization of a Computer for the Iterative Solution of PDE's," unpublished manuscript, Princeton University, 1971.

[6] Paker, Y., "Application of Microprocessor Networks for the Solution of Diffusion Equation," *Mathematics and Computers in Simulation*, Volume 19, No. 1, March 1977.

[7] Doenin, V. V.,"Parallel Digital Network Processor and Transient Stability Analysis in the Processor's Logical Network," *Avtomatika i Telemekhanika*, (in translation), Vol. 40, No. 8, August 1979, pp. 139-149.

[8] W. F. Ames, *Numerical Methods for Partial Differential Equations*, Barnes & Noble, New York, 1969.

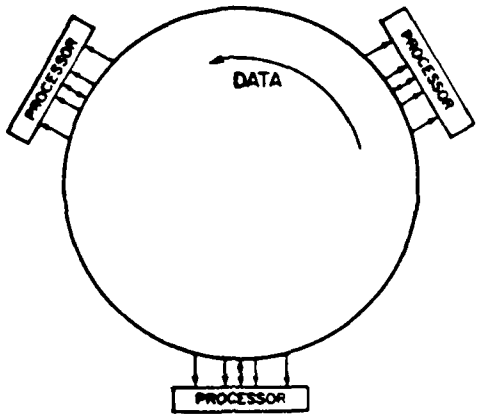


Fig. 1 Circulating store configuration.

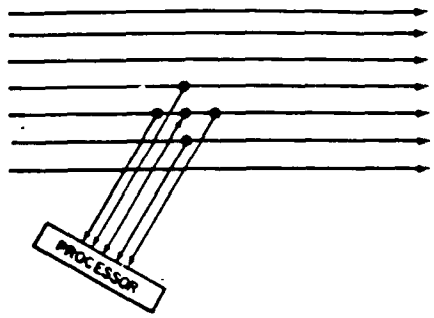


Fig. 2 Data access of one processor in a 5-point iteration for a two-dimensional problem.

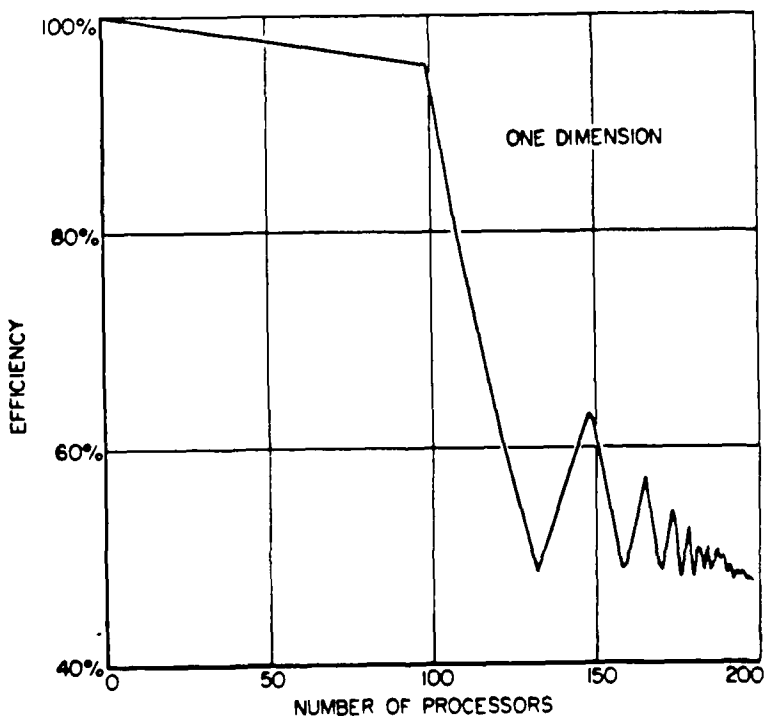


Fig. 3 The efficiency vs. number of processors for a one-dimensional problem; 200 grid points.

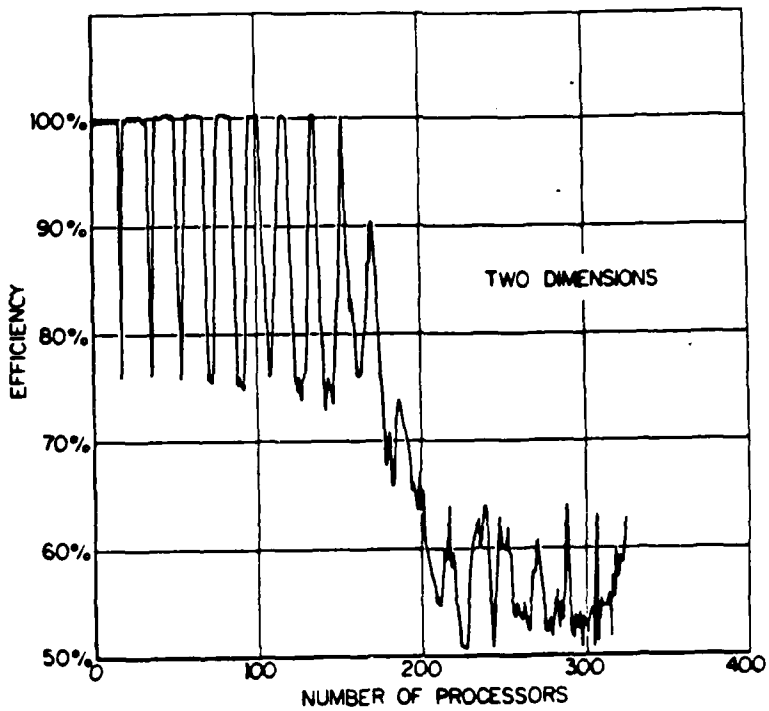


Fig. 4 The same as Fig. 3, for a 20x20 two-dimensional problem.

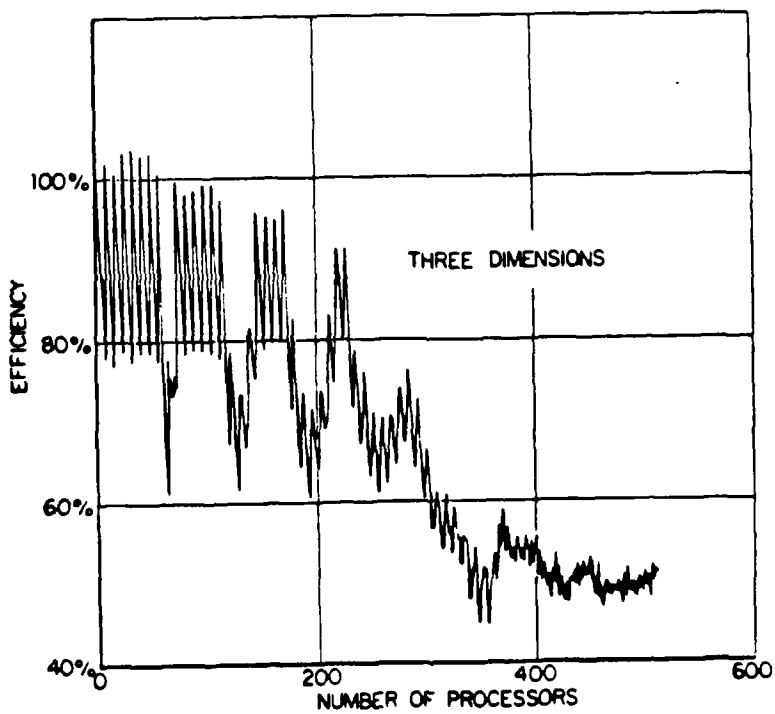


Fig 5 The same as Fig. 3, for a 10x10x10 three-dimensional problem.

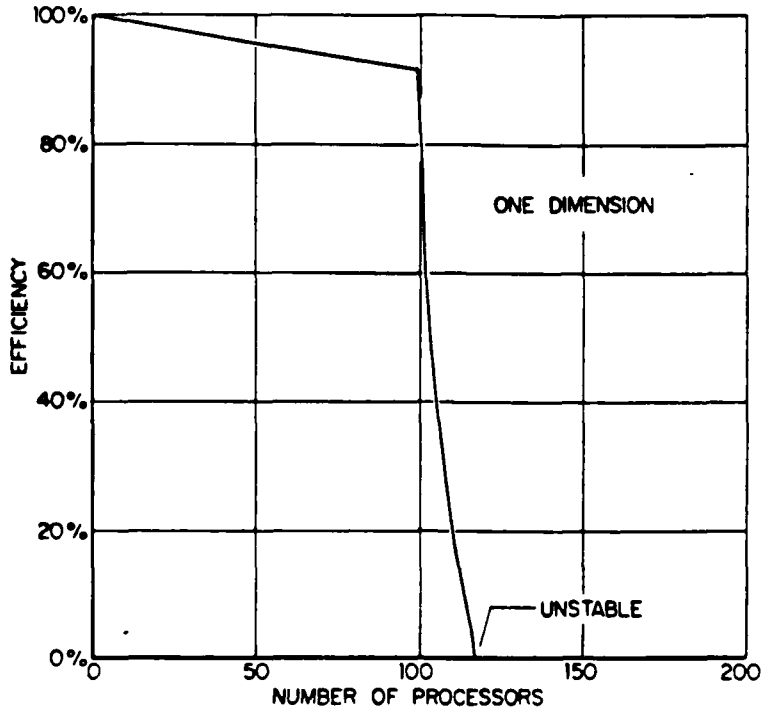


Fig. 6 The same as Fig. 3 for the over-relaxation method with $\alpha = 1.5$.

Optimal Choice of Intermediate Latching to Maximize Throughput in VLSI Circuits

PETER R. CAPPELLO, MEMBER, IEEE, ANDREA LaPAUGH, MEMBER, IEEE, AND KENNETH STEIGLITZ, FELLOW, IEEE

Abstract—In many computational tasks, especially in signal processing, it is the throughput that is important, rather than the latency, or delay. If a special-purpose VLSI chip is designed for a particular signal processing task, such as FIR filtering, for example, the maximum clock rate, and hence throughput, is determined by the depth of the combinational logic between registers and the time required for the distribution and operation of the clock. If the combinational logic is sufficiently deep (in bit-parallel circuits, for example), the throughput can be increased by inserting intermediate stages of clocked latches. This is at the expense of increased area and delay to operate and clock the intermediate registers. Roughly speaking, the strategy amounts to using more of the chip area to store information useful for pipelining.

This paper investigates the optimal tradeoff between the degree of intermediate latching and cost, using the measure AP, where A is the chip area and P is the period (the reciprocal of throughput). We derive expressions for the time and area before and after intermediate latching, using the Mead-Conway model, both for the cases of on-chip and off-chip clock drivers. The results show that significant reductions in AP product (reciprocal of throughput per unit area) can be achieved by intermediate latching in many typical signal processing applications, for a wide range of circuit parameters. The array multiplier is used as an example

I. Introduction

WHEN certain tasks are implemented with special-purpose VLSI chips, it is often the *period* P (time between successive outputs) that is crucial, rather than the latency or delay T. This is especially true in signal processing, where typical tasks such as filtering and discrete Fourier transformation often have high volume requirements and relatively lax delay requirements. Recent work has described bit-serial and bit-parallel VLSI architectures that do in fact allow the period to be equal to the clock period (see, for example, [2], [4]-[9], [12]). In [5], [7] a class of these circuits is called *completely pipelined*. In this paper, we take up a different question, that of inserting intermediate stages of latching so as to maximize the rate at which the clock can run without a disproportionate blowup in area requirements. We will use the criterion of minimizing the AP product, where A is the area of the VLSI circuit and P is the period. The AP product can be thought of as the reciprocal of throughput per unit area, and a completely pipelined circuit optimal with respect to this criterion can be claimed to make best use of chip area. Leiserson and Saxe [14] treat the related problem of redistributing latches so as to decrease period, but they do not consider area or clocking penalities.

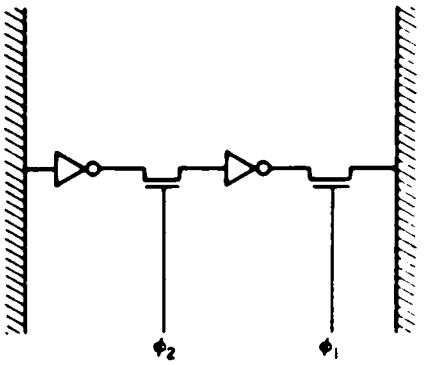


Fig. 1. Two-phase clocked latches between stages of combinational logic.

We assume that the circuits we discuss are designed along the lines described by Mead and Conway [1]: typically that a two-phase clock is used to transfer information between *registers* (or *latches*), and that these registers are separated by combinational logic. The following sections are devoted to modeling the time and area requirements of the latches, the combinational logic, and the clock driver. We then consider the overall circuit and investigate the optimal choice of the amount of latching for the two cases of on-chip and off-chip clock drivers. While the assumptions made about first-order circuit behavior pertain to *n*MOS technology, the analysis technique uses dimensionless parameterization and is applicable to any situations with deep combinational logic—typically bit-parallel circuits. A representative tradeoff curve is shown for an example.

II. Clock Timing

We will adopt a version of the two-phase clocking system described by Seitz in [1, ch. 7], a typical stage of which is shown in Fig. 1. Fig. 2 shows the corresponding timing diagram: First, we must drive the phase 1 clock signal ϕ_1 high, taking time t_{clock} (the *clock driver* time). We then need a minimum time t_{delay} (the delay time) to charge the input stage of the combinational logic. Phase 1 must then go low (taking time t_{clock}), and phase 2 must then go high (also taking time t_{clock}). We must insure that there is a minimum time t_{12} dur-

Manuscript received August 10, 1982; revised April 12, 1983. This work was supported in part by the National Science Foundation under Grant ECS-8120037, U.S. Army Research-Durham under Grant DAAG29-82-K-0095, and DARPA Contract N00014-82-K-0549. A preliminary version of this paper was presented at the 1983 IEEE International Conference on Acoustics, Speech, and Signal Processing, Boston, MA, April 14-16, 1983.

P. R. Cappello was with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08544. He is now with the Department of Computer Science, University of California, Santa Barbara, CA 93106.

A. LaPaugh and K. Steiglitz are with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08554.

0096-3518/84/0200-0028$01.00 © 1984 IEEE

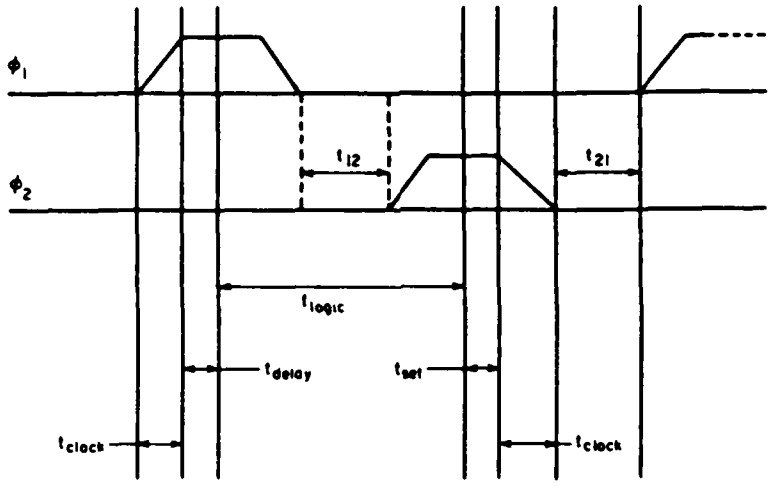


Fig. 2. Clock-timing diagram.

ing which both clocks are low; otherwise we run the risk that skew between the clock phases will cause both clocks to be on at the same time. This brings us up to the point where the combinational logic has already started to work.

The input values propagate through the combinational logic, taking some time t_{logic}. This time includes the time during which ϕ_1 is brought down and ϕ_2 is brought up. The time t_{logic} will ordinarily dominate the clock-interchange time, but, in general, we need to set the time for this operation to

$$t = \max (t_{logic}, 2t_{clock} + t_{12})$$

where, for safe operation of the circuit, t_{logic} must of course be taken as the *maximum* delay time of the combinational logic.

We next need to transfer the output values of the preceding logic stage to the input of the latch whose output is controlled by ϕ_1: that is, ϕ_2 must remain on for a minimum charging time t_{set} (the *preset* time). The ϕ_2 clock signal must then be brought down (taking another clock driver time t_{clock}, and another dead time (t_{21}) provided to insure nonoverlap of clocks in case of clock skew.

The minimum period P of the circuit is therefore

$$P = 2t_{clock} + t_{delay} + t_{set} + t_{21} + \max (t_{logic}, 2t_{clock} + t_{12}).$$

To be more accurate, we might want to take into account the fact that the upgoing and downgoing clock waveforms are not completely symmetric; but the term t_{clock} can be taken to represent the average of the upgoing and downgoing clock times in a single driver. In a multistage driver the stages alternate up and down, and we can take t_{clock} to be the sum of the averages of the upgoing and downgoing times along the driving chain.

III. Latch Time and Space

We next want to express the time delay of the latches in terms of basic units that are determined by the technology. For this purpose, we consider the *n*MOS inverter with a minimum size pulldown and a pullup/pulldown ratio of 4 to be the *basic* cell, with area A, pulldown gate capacitance C, effective pulldown resistance R, and pulldown time (*transit time*) τ when driving the input of an equal size inverter. We refer to such a cell in what follows as a *minimal inverter*.

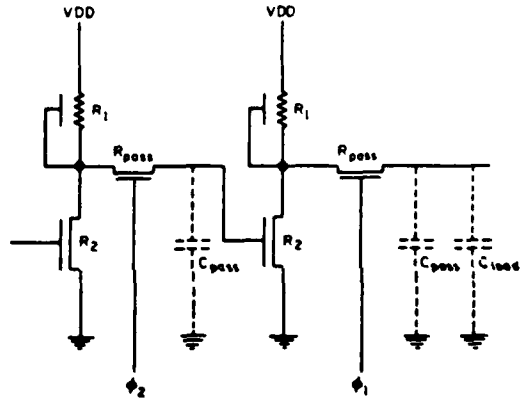


Fig. 3. Details of the clocked latches, showing pullup and pulldown effective resistances and capacitances.

Now inverters in the latches are driven through pass transistors, so the discussion in [1] shows that we should choose a pullup/pulldown ratio of 8. The time required for the second inverter to charge its load is therefore approximated by the following RC constant:

$$t_{delay} = (R_1 + R_{pass})(C_{load} + C_{pass})$$

where the R's and C's are shown in Fig. 3. Assuming that the pass transistors are minimum size, $R_{pass} = R$ and $C_{pass} = C$. Also assuming that the capacitive load (input to the combinational logic) is minimal, we get

$$t_{delay} = 2(R_1/R + 1)\tau$$
$$= 2(L_1/W_1 + 1)\tau$$

where, from now on, we express resistance in terms of the length-to-width ratio of the transistor

$$R_1 = (L_1/W_1)R.$$

If the pullup/pulldown ratio of the latches is taken to be 8

(as mentioned above), we can write the normalized delay time as

$$t_{delay}/\tau = 2(8r + 1)$$

where $r = L_2/W_2$ is the size of the latch pulldown. When $r = \frac{1}{2}$ the pulldown transistor of the latch inverter will be twice as wide as the corresponding transistor of the minimal inverter, but the pullup/pulldown ratio is 8, not 4, so the pullup transistor will then be the same length as in the minimal inverter. The area of such a latch inverter with $r = \frac{1}{2}$ will be only a little larger than that of a minimal inverter, perhaps about 25 percent larger. Thus, the choice of $r = \frac{1}{2}$ speeds up the latch without much area penalty, and we will use this value in this paper, although it could be kept as a parameter.

Using a similar argument based on RC charging times, the preset time is

$$t_{set}/\tau = (8r + 1)(1/r + 1).$$

The $1/r$ term comes from the input capacitance of the second inverter, which loads the first inverter. To see this, write

$$C_{load} = (L_2 W_2/L W) C = (W_2/L_2) C = (1/r) C$$

where $L_2 = L = W$ are minimum size.

The latching area is easy to write down. Assuming that the pass transistors are the same size as minimal inverters, and that the latches have area $1.25A$, each two-phase latch requires normalized area

$$A_{latch}/A = 2(1.25 + 1) = 4.5.$$

IV. Combinational Logic Time and Space

We want a fairly general model for the combinational logic that is sandwiched between the latches: such logic may be built from NAND and NOR gates, pass transistors, or some combination of the two. We will assume that the typical logic stage is a uniform array of $n \times k$ *logical elements*, each of which has an area A_{elem} and a delay τ_{elem}, where

$$A_{elem} = \alpha A$$

and

$$\tau_{elem} = \beta\tau$$

This array will be thought of as n rows by k columns, with a maximum delay path from left to right of k elements. Since logic stages are not usually so uniform, the α and β parameters must represent *average* values for the combinational logic. If gates are built out of inverters and coupled directly, for example, β will generally be determined by the fan-out factor of the logic and the size of the inverters. An average fan-out factor of 3, using gates (with a pullup/pulldown ratio of 4), will result in $\beta \approx 12$, because we must allow for the worst case in the propagation of logic, where all signals are upgoing. To reduce this to a value closer to that of a minimal inverter, we expect to increase the area to, say, twice that of a minimal inverter. Thus, we can take values of $\alpha = 2$ and $\beta = 4$–12 as typical of combinational logic implemented with arrays of gates. We should also note that the value of α should be selected to reflect the space per logical element required for power and ground lines.

We will assume that the nominal circuit has one typical logic stage between a pair of two-phase latches, and we then consider the insertion of $(m - 1)$ latches equally spaced in the combinational logic, $m \geq 1$. The case $m = 1$ then represents the original situation. We assume the latches can be made to "fit" well; that is, that the combinational logic is arranged regularly enough so that stages can be pushed apart and columns of latches inserted. The total time required for the logic is therefore

$$t_{logic}/\tau = \beta(k/m)$$

and the area

$$A_{logic}/A = \alpha d k^2$$

where $d = n/k$ is the height-to-width ratio of the original logic block, another dimensionless parameter, usually assumed to be 1.

V. On-Chip Clock Driver Time and Space

If we use an on-chip clock driver, we want to use a multistage version as described in [1], since the driver will have a large capacitative load, especially if there is an appreciable amount of intermediate latching introduced. We assume that clock distribution is on metal, so that propagation delay along the wires is small. Each stage is assumed to have a pulldown f times the size of the preceding, so if there are S stages driving Y pass transistors, each with minimal capacitance C,

$$f = Y^{1/S}.$$

If we start the clock driving with a minimal inverter, the normalized delay of such a driver is approximately

$$t_{drive}/\tau = 2.5fS.$$

The factor of 2.5 results from averaging the pullup time of 4τ and pulldown time τ along the inverter chain. (If we do not insist that S is an integer, and we minimize this delay with respect to f, we get the value $f = e$ [1]. But S *is* an integer.)

This estimate for delay assumes that we insist on a globally-synchronized clock—that the clock signals at the input of the driver can be used anywhere else without concern for synchronization. Caraiscos and Liu [11] have pointed out that the rise and fall times of the clock waveforms may be much smaller than the absolute delay, and that using a local clock may allow higher throughput, at the expense of using local clock signals that must be made synchronous with the signal itself at different points on and off the chip. Sending the clock along with the signal will incur other costs, of course. (For a discussion of the virtues of a globally-synchronized clock in signal processing, see [10]). The analysis in this paper is conservative in the sense that the resulting degree of latching and increase in throughput is on the low side. (We can avoid the area and delay penalty incurred by using an on-chip driver by moving the clock driver off-chip. That case will be discussed in more detail in Section VII.)

We must also consider the area contribution of the clock driver in relation to the rest of the circuit. The normalized area of the driver is

$$A_{\text{drive}}/A = \sum_{i=0}^{S-1} f^i = (Y - 1)/(f - 1).$$

Next we look at the overall time and space requirements of the circuit.

VI. Optimization of *AP* Product with an On-Chip Clock Driver

We can now write the total minimum normalized period $P/\tau = p$ in terms of our parameters as follows:

$$p = 5fS + 25 + \tau_{21} + \max(\beta k/m, 5fS + \tau_{12})$$

where, as above,

$$f^S = Y = (m + 1)\,n = \text{number of lines driven}$$

and $\tau_{12} = t_{12}/\tau$, $\tau_{21} = t_{21}/\tau$. Similarly, the total normalized area $area/A = a$ is

$$a = 2(Y - 1)/(f - 1) + 4.5Y + \alpha kn$$

where the factor of 2 accounts for the fact that we must have two drivers, one for each phase. (These can be combined to some extent, but the total area is still nearly twice that of a single driver.)

We now have the function $ap(m, S)$, where m and S are discrete parameters. The number of stages is never much larger than ln Y, since the optimal choice of f is usually around e. In most cases of interest, therefore, it suffices to take the minimum of ap for $S = 1, \cdots, 16$, producing what we call $ap(m, *)$:

$$ap(m, *) = \min_S ap(m, S).$$

The range of m is certainly between 1 and k, so the optimal choice of m can be determined simply by

$$ap(*, *) = \min_m ap(m, *).$$

The *gain* G in *AP* product achieved by latching is, therefore,

$$G = ap(1, *)/ap(*, *).$$

VII. The Case of an Off-Chip Clock Driver

As mentioned in Section V, if we allow the clock driver to be off-chip, we can drive the larger capacitive loads incurred by extra latching with essentially no penalty in clock delay or driver area. The normalized period and area can then be written

$$p = 2\tau_{\text{clock}} + 25. + \tau_{21} + \max(\beta k/m, 2\tau_{\text{clock}} + \tau_{12})$$

$$a = 4.5Y + \alpha kn$$

where we have assumed some delay of $\tau_{\text{clock}} = t_{\text{clock}}/\tau$ for the clock rise and fall times. The ap product is therefore a function of only one unknown parameter, m.

With these changes in a and p, the same methodology applies—a numerical example will be given in the next section. Note, however, that now the optimal value of m will occur roughly near the breakpoint where $\beta k/m = 2\tau_{\text{clock}} + \tau_{12}$, and that these times are both highly uncertain and small in size. The analysis in this case is therefore much less reliable, and much more sensitive to unmodeled effects such as propagation delay, than in the on-chip clock driver case.

VIII. Numerical Examples

We now give some typical numerical results. For this purpose, we consider a 16-bit array multiplier, implemented by an array of full adders, as described, for example, in [2]. We also assume that the full adders are implemented with gates: each full adder will then be about 3 gates deep. The carry propagation will require an array that has a maximum depth of 2×16, so altogether the combinational logic will have $k \approx 100$. (This is consistent with the value of "113 gate delays" given in [3].) Say that each gate takes about double the area of a minimal inverter ($\alpha \approx 2$, optimistic for area, and hence pessimistic for our purposes), and that, as discussed in Section IV, $\beta \approx 6$. The array is roughly square, so that $d \approx 1$. Finally, we will assume that clock skew is not an important problem, and take $\tau_{12} = \tau_{21} = 4$.

Fig. 4 shows a plot of normalized period $p(m, *)/p(1, *)$; normalized area $a(m, *)/a(1, *)$; and normalized *AP* product $ap(m, *)/ap(1, *)$ versus m. The period as a function of m decreases sharply (roughly as $1/m$) until the combinational logic time is dominated by the clock-swapping and dead time (that is, until $t_{\text{logic}} \approx 2t_{\text{clock}} + t_{12}$). After this point the clock-driving time will determine the minimum clock period and it no longer pays to increase m, because the area will increase with no payoff in speed. The minimum value of period occurs close to the minimum value of *AP* product. Thus, in theory, the period can be decreased somewhat from its value when the *AP* product is minimized, at a slight cost in area. In practice the optimal values are almost always nearly equal, and sometimes identical, because of the discreteness of the parameters m and S.

Fig. 5 shows a plot of gain G in *AP* product versus the depth of combinational logic k, for the values $\alpha = 2$ and $\beta = 4, 6, 8, 12$. The graph shows significant gains in *AP* product (more than 2) over the unlatched case when $k \geq 50$ and $\beta \geq 6$. Even when the gates are as fast as a minimal inverter (worst-case delay factor $\beta = 4$), there is an *AP* product gain of 2.2 when $k = 100$. Note that a larger value of α would only improve the gain.

We conclude by looking at the actual numerical values of the minimum clock periods and areas involved in this analysis. Taking the $k = 100$, $\alpha = 2$, $\beta = 6$ case above for a hypothetical 16-bit array multiplier, and assuming $\tau = 0.3$ ns for current technology, we get a period of $P = 210$ ns with no intermediate latching, and an optimal period of $P = 66$ ns with $m = 6$ (5 intermediate latching stages).

The area before latching is $2.11 \times 10^4 A$, which at $\lambda = 1.5\mu$ (3μ line width) and a $225\lambda^2$ inverter is about 10.7 mm^2. After the intermediate latching, the area becomes 12.1 mm^2; certainly a modest increase in area for about a threefold increase in speed.

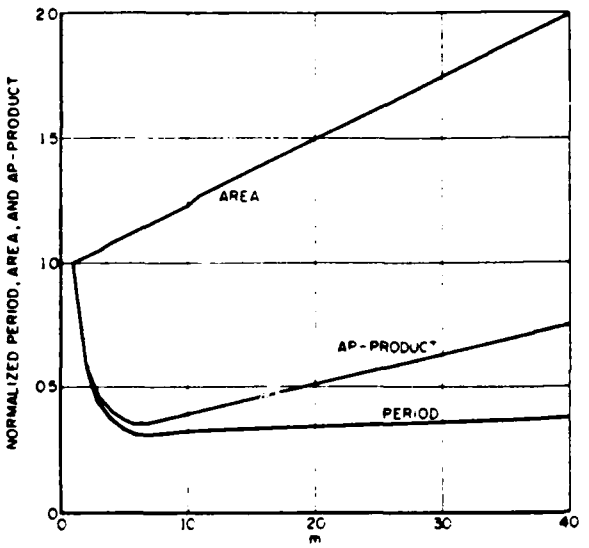


Fig. 4. Normalized period, area, and AP product versus m for $\alpha = 2$, $\beta = 6$, $k = 100$. The parameter $(m - 1)$ is the number of intermediate latching stages.

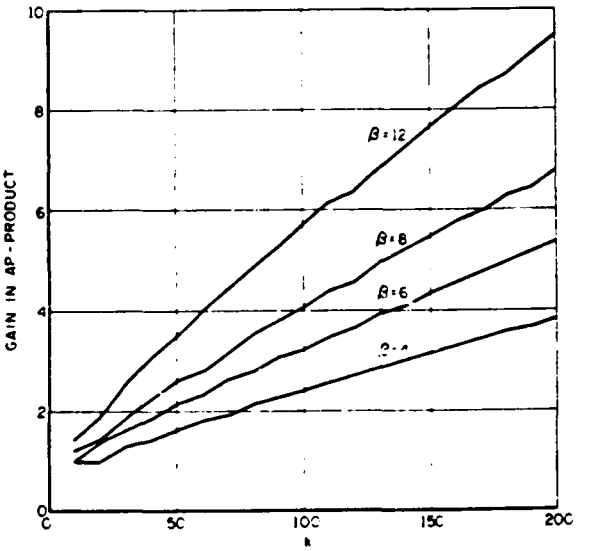


Fig. 5. Gain in AP product versus combinational logic depth k for $\beta = 4, 6, 8, 12$. The parameter β is the delay of a combinational logic element, normalized in terms of that of a minimal inverter.

The preceding example assumed an on-chip clock driver. When we use an off-chip clock driver at presumed small cost, as discussed in Section VI, we naturally get much faster solutions. In this example, the optimal value of period with the parameters of Section VII and $\tau_{clock} = 4$ (assuming a very sharp clock rise time and fall time), minimizing AP product, is 18 ns, compared with the unlatched value of 191 ns. The area goes from 10.6 mm^2 with no latching to 16.5 mm^2 with latching. This large increase in area reflects a corresponding increase in the density of latching: 26 ($m = 27$) latching stages are introduced. We emphasize that in the case of an off-chip clock driver, the numerical values of the parameters t_{12} and t_{clock} are very uncertain and the optimal values of period, area, and latching density are sensitive to these parameters. The large predicted speedups in possible clock rate may not be realizable in practice.

IX. Conclusions

We have modeled the timing of a generic pipelinable VLSI circuit in which there are combinational logic stages separated by latching stages driven by two-phase clocks. An array multiplier is typical of such a configuration. We then investigated the effect of introducing intermediate latching stages, especially the tradeoff between increased throughput and increased area. Expressions were derived for area and minimum clock period, normalized in terms of minimal inverter area and delay, and we showed that optimal choices of the number of clock driver stages (S), and the number of intermediate latching stages ($m - 1$), can be made by simple enumeration.

The numerical results illustrate the choice of latching density in a typical signal processing application. According to our model, a 16-bit array multiplier with gate logic and an on-chip multistage clock driver can be clocked about three times faster with about a 13 percent increase in area using five intermediate latching stages. This decrease in period is also accompanied by an increase in the latency, or delay, of the multiplier.

Higher throughput can be achieved with an off-chip clock driver, but the parameters in that case are less well known, and at such speeds the model becomes less reliable.

Much more work needs to be done on detailed modeling of the timing of such VLSI circuits if we are to achieve maximum throughput rates in applications like signal processing. Future work will attempt to refine our model, along the lines of [13] as an example. We also need to study propagation delay, which was assumed to be relatively small in the examples (4 times the minimal inverter gate delay τ for clock distribution, a reasonable assumption if the clock lines are metal, for example). Another important set of interesting problems concerns the study of the way algorithms, topologies, and layouts interact with the timing problems considered here. Recent work on completely pipelined or bit-level systolic arrays is a start in that direction (see, for example, [2], [4]-[9], [12]).

Acknowledgment

We are indebted to C. Caraiscos and B. Liu for valuable comments on the manuscript.

References

[1] C. Mead and L. Conway, *Introduction to VLSI Systems*. Menlo Park, CA: Addison-Wesley, 1980.

[2] J. V. McCanny, J. C. McWhirter, J. B. G. Roberts, D. J. Day, and T. L. Thorp, "Bit level systolic arrays," in *Proc. 15th Asilomar Conf. Circuits, Syst., Comput.*, Nov. 1981.

[3] K. Bötcher, A. Lacroix, M. Talmi, and D. Wesseling, "Integrated floating point signal processor," in *Proc. 1982 IEEE Int. Conf. Acoust., Speech, Signal Processing*, Paris, France, May 1982, pp. 1088-1091.

[4] P. R. Cappello, and K. Steiglitz, "Digital signal processing applications of systolic algorithms," *CMU Conf. VLSI Syst. Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds. Rockville, MD: Computer Science Press, 1981.

[5] —, "Bit-level fixed-flow architectures for signal processing," in *Proc. 1982 IEEE Int. Conf. Circuits, Comput.*, Sep. 29-Oct. 1, 1982.

[6] —, "A VLSI layout for a pipelined dadda multiplier," *ACM Trans. Comput. Syst.*, vol. 1, May 1983.

[7] —, "Completely pipelined architectures for digital signal pro-

cessing," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-31, pp. 1016–1022, Aug. 1983.

[8] H. T. Kung, L. M. Ruane, and D. W. L. Yen, "A two-level pipelined systolic array for convolutions," *CMU Conf. Syst. Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds. Rockville, MD: Computer Science Press, 1981.

[9] P. B. Denyer and D. J. Myers, "Carry-save arrays for VLSI signal processing," in *VLSI 81: Very Large Scale Integration*, J. P. Gray, Ed. London: Academic, 1981.

[10] R. F. Lyon, "A bit-serial VLSI architecture methodology for signal processing," in *VLSI 81: Very Large Scale Integration*, J. P. Gray, Ed. London: Academic, 1981.

[11] C. Caraiscos and B. Liu, private communication.

[12] —, "Bit serial VLSI implementations of FIR and IIR digital filters," in *Proc. 1983 Int. Symp. Circuits Syst.*, May 1983.

[13] P. Penfield, Jr. and J. Rubinstein, "Signal delay in RC tree networks," in *Proc. Second California Inst. Technol. Conf. VLSI*, 1981.

[14] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," in *Proc. 22nd Ann. Symp. Foundations of Comput. Sci.*, October 28-30, 1981.

Peter R. Cappello (M'83) was born in Queens, NY, on October 18, 1948. He received the B.S. degrees in mathematics and in computer science from Pennsylvania State University, University Park, in 1970, the M.S. degree in electrical engineering and computer science from the University of California, Berkeley, in 1973 (while a member of the Technical Staff of Bell Laboratories), and the Ph.D. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1982.

He is now with the Department of Computer Science, University of California, Santa Barbara, CA.

Andrea LaPaugh (M'81) was born in Middletown, CT, on June 26, 1952. She received the A.B. degree in physics from Cornell University, Ithaca, NY, in 1974, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1977 and 1980, respectively.

She subsequently spent a year as a Visiting Assistant Professor in the Department of Computer Science, Brown University, Providence, RI. Since September 1981 she has been an Assistant Professor in the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ.

Dr. LaPaugh is a member of the IEEE Computer Society and the Association for Computing Machinery.

Kenneth Steiglitz (S'57–M'64–SM'79–F'81) was born in Weehawken, NJ, on January 30, 1939. He received the B.E.E., M.E.E., and Eng.Sc.D. degrees from New York University, New York, NY, in 1959, 1960, and 1963, respectively.

Since September 1963 he has been with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, where he is now Professor, teaching and conducting research in the computer and systems areas. He is the author of *Introduction to Discrete Systems* (New York: Wiley, 1974), and coauthor, with C. H. Papadimitriou, of *Combinatorial Optimization: Algorithms and Complexity* (Englewood Cliffs, NJ: Prentice-Hall, 1982).

Dr. Steiglitz has served as a member of the Digital Signal Processing Committee of the IEEE Acoustics, Speech, and Signal Processing Society, and as an Administrative Committee member and Awards Chairman of the Society. He is Associate Editor of the journal *Networks*. He is a member of Eta Kappa Nu, Tau Beta Pi, and Sigma Xi, and in 1981 received the Technical Achievement Award of the ASSP Society.

M^3 Notes

R. E. Cullingford
H. Garcia-Molina
P. Honeyman
R. J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08544

1. Overview

This is a short collection of notes on the latest results from the Massive Memory Machine (M^3) group. Most of the notes concern the magnitude of the speedup possible with massive amounts of physical memory. We are greatly encouraged by the results recently obtained, and are of course eager to see a real M^3 in operation soon.

In addition, there is growing industrial interest and support for the M^3 concept. As we reported earlier, DEC is very enthusiastic about working with us on a large memory VAX, as well as on the ESP architecture. Furthermore, our friends at DEC have just told us that DEC will soon be announcing actual product VAX's with 128Mb of memory. While this is not the 256-512Mb we are planning, it is exciting to see that they are thinking along similar lines.

There are also two new groups that are interested in M^3. The first is a group at Bell Labs at Murray Hill. They would like to build an M^3 to solve certain phone company transaction problems that very high speed transaction rates. These could easily be accomplished on a 1 MIPS M^3; but would require a huge number of parallel processors if the data were stored on rotating disks. They are

working with us on plans for a small prototype.

The second group is at IBM Yorktown. They are quite interested in the whole M^3 concept; they found out about M^3 by reading our recent IEEE publication. We have just met with Dr. Frank Moss, the project leader, and we are planning a joint two day meeting in a few weeks in Princeton. We hope that through such meetings we can work out a strategy for formal cooperation.

1.1. PROLOG Studies

PROLOG is widely touted as the language of choice for expert systems research and development. Consequently, we view PROLOG as a solid basis for experiments in applications of massive memory. The early returns are most encouraging: we are discovering general techniques for speeding up PROLOG programs, as well as a number of tricks that we can apply in special circumstances.

1.2. Program Tracing

In order to better understand the data reference patters of memory intensive programs, we have implemented a software tracing package. It is being used to study several programs, including the Clay solver, and to predict their running time on machines of various memory sizes and architectures. The trace package uses the UNIX debugging facility to interrupt the program under analysis after each instruction execution. When it is interrupted, the data location(s) being accessed are recorded.

The main problem with this package is that is slows down program execution considerably, roughly by a factor of 3000. To alleviate this problem, we are implementing a VAX simulator capable of producing the same trace information. Preliminary experiments with the simulator indicate that it is 10 times faster than the original package. Although this still represents substantial overhead, the new simulator will let us study a wider range of programs.

1.3. ESP Straw Man Prototype

We have started implementing a preliminary version of a ESP machine. The goal is not to create an operational system, but to gain experience with the ESP architecture and to identify some of the practical problems that may arise in a full implementation.

We have acquired two 8086-based microprocessor systems. Each CPU talks to its local memory via a multibus. Each CPU is also connected to a pair of disk drives. We have designed and wire-wrapped two simple ESP controllers; each controller sits on one of the multibuses. The controllers are tied together by a simple broadcast bus. The controllers make no provisions for failures or errors.

We have started debugging the ESP controller hardware, and are only beginning to design the software that will run the machines. Even at this early stage, our implementation effort has already turned up several important issues that had been overlooked in the original paper design. These issues include system startup, I/O and interrupt handling, periodically refreshing dynamic memory, and queueing data words at each ESP controller; we are now studying these issues. Some of them can be safely ignored in our prototype (e.g., memory refresh); to cope with others, we are adding more capabilities to our ESP controllers.

1.4. M^3 Performance on Database Benchmarks

Two recent papers have compared the performance of several database machines, and we decided to evaluate the performance of an M^3 database machine on the same queries that were used for the benchmarks. Our preliminary results are given in an attached report.

In summary, our results clearly show that an M^3 that can hold all of the database in fast memory can outperform the database machines considered. The speedups range from a factor of 7, to a factor of 27,000, depending on the assumptions made and the sample queries analyzed. These results must of course be treated with caution, but they do illustrate that memory can be an extremely useful resource for database applications.

2. Massive Memory vs. Massive Parallelism

2.1. Introduction

A common "folk principle" is that massive parallelism is the only way to vastly speed up computations. In contrast, we will show that there are important classes of computations which can be greatly sped up only by massive amounts of physical memory. Thus, for these computations an $\mathbf{M}^3$ will vastly outperform any parallel machine!

On the face of it our claim seems absurd. Don't parallel machines always dominate sequential machines such as an $\mathbf{M}^3$? In order to understand this apparent paradox let us examine the standard argument more carefully. Assume that some task has an algorithm A that takes time $T(A,n)$ for inputs of length n. Then potentially p parallel processors can run this algorithm in time $T(A,n)/p$. Of course this is the upper limit on the potential performance of p parallel processors; in practice fully linear speed up is rare. However, to make our point about the power of memory over parallel processor even more dramatic, let's assume that such speedup is always possible.

Since $T(A,n) \geq T(A,n)/p$ for any p, how can parallel processors ever lose to a sequential machine such as $\mathbf{M}^3$? The answer is that there may be a new algorithm B for which $T(B,n) < T(A,n)/p$ for any reasonable p. Moreover, this algorithm may require in an essential way vast amounts of random access memory; thus, this algorithm *cannot* be executed on the p parallel processors for lack of physical memory. In this way it is possible for an $\mathbf{M}^3$ to greatly outperform any collection of parallel processors. Note, we are not saying that memory is always better than processors, this is false; but then so is the folk principle that parallel processors are always better than sequential machines. We are simply pointing out that there are memory intensive computations that benefit much more from memory than from processors.

A possible counter to our argument is: why can't the parallel processors have enough space to use the better algorithm? Of course in principle they can. The key point is that on many interesting problems we will not be able to afford *both* parallel processors *and* massive amounts of memory. On problems that

fundamentally require memory, not processors, the parallel processors will be forced to run a slower algorithm, and hence be outperformed by an $\mathbf{M}^3$.

We now demonstrate our claims with two examples from PROLOG, an important language for a wide variety of non-numeric computations. There is currently an intensive international effort to use parallelism to speed up PROLOG. It may therefore be interesting to see how massive amounts of memory can be used to achieve vast speedups in PROLOG.

2.2. Recursion

The first application of memory is conveniently introduced by way of a simple PROLOG example:

```
path([A|X],[B|Y]) :- edge(A,B).
path([-|X],Y) :- path(X,Y).
path(X,[-|Y]) :- path(X,Y).
```

(Here edge(,) is some relation that is defined by other rules.) Path(X,Y) checks the two lists X and Y to see if there is an element in the first list with an edge to an element in the second list. Intuitively, we would expect that this process should take time quadratic in n, the total number of elements. However, on any standard PROLOG it takes exponential time, because PROLOG repeatedly re-evaluates subgoals. While there are at most quadratically many subgoals, they are evaluated exponentially many times.

For those unfamiliar with PROLOG's evaluation scheme, let us examine the computation of path(X,Y) in more detail. Here X is a list $x_1, \ldots, x_k$ and Y is a list $y_1, \ldots, y_l$ with $k+l=n$. Path(X,Y) is computed as follows:

(1) If, either list is empty then path(X,Y) is false.

(2) Next, if edge(x_1, y_1) is true then path(X,Y) is true.

(3) Finally, if either path(X',Y) or path(X,Y') are true then so is path(X,Y). Here X' is equal to $x_2, \ldots, x_k$ and Y' is equal to $y_2, \ldots, y_l$.

Note, the last part of the computation is the key to the use of repeated subgoals. The call to path(U,V) where U is equal to $x_i, \ldots, x_k$ and V is equal to $y_j, \ldots, y_l$ occurs exactly $\binom{i+j-2}{i-1}$ times.

Let us now compare the performance of a set of p parallel processors on this example and an $\mathbf{M}^3$. The p parallel processors take $2^n/p$ time since the usual PROLOG implementation checks that many subgoals on this problem. On the other hand, an $\mathbf{M}^3$ can use the following strategy: cache all subgoals and use table lookup instead of re-evaluating subgoals. This strategy leads to an algorithm that takes order n^2 time, since each subgoal is checked exactly once. Thus, even for modest sized problems (n equal to 40) the number of parallel processors required to perform as well as the $\mathbf{M}^3$ is on the order of one billion!

A final word about this example: it is of course always possible to create examples that make any approach look good. We feel, however, that using memory to avoid repeated re-evaluation of subgoals is a fundamental technique to speed up PROLOG. Exponential growth cannot simply be waved away: there are many natural PROLOG examples that lead to the same combinatorial explosion. A PROLOG machine with a huge memory to cache millions or even billions of subgoals would be extremely powerful.

2.3. Table Lookup

A second critical use of memory to speed up PROLOG relies on the way the PROLOG data base is searched. In order to reach a goal PROLOG searches its rules for the first one that matches the current goal. While there are a number of ways to speedup this search, the fastest one appears to use large amounts of extra memory. The idea is simple: in additional to storing the rules, we store indices (inverted lists) that make the search very fast. With the proper data structures a constant time search *independent* of the size of the data base is possible. Clearly, no number of parallel processors could outperform such an implementation.

We have performed a number of experiments to validate this claim. Our experiments so far have consisted of comparing the standard implementation of PROLOG with ones that use the data structures described above. One test program is a simple PROLOG program that computes the transitive closure of a directed graph:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- edge(X,Z), reach(Z,Y).
```

(Again, edge(,) is a relation that is defined by other rules.) Table 1 contains the actual results of experiments on a VAX 11/750. The speedups are dramatic: even on modest sized graphs we get several orders of magnitude speedup. The reason for these large speedups is that the parallel approach takes order n^2/p time and the memory intensive $\mathbf{M}^3$ approach takes only order n time. Since n reflects the size of the data base, the potential for speedups large data bases is immense.

Number of Edges	Number of Queries	PROLOG (secs)	$\mathbf{M}^3$ - PROLOG (secs)
78	156	224.3	0.5
60	380	252.1	1.0
100	380	3525.7	1.2
100	870	3700.8	2.8
120	1190	18375.8	4.8
100	2450	6450.6	9.8
165	2450	?	12.2

TABLE 1

Results of comparison of PROLOG and $\mathbf{M}^3$-based PROLOG implementation.
All times in VAX 11/750 seconds.

3. M^3 Performance on Certain Database Benchmarks

In recent papers by Hawthorn and DeWitt [1], and by Hillyer, Shaw, and Nigam [2], the performance of several database machines was compared. In this note, we study the performance of an M^3 database machine on the same queries under comparable assumptions.

A M^3 is not a "conventional" database machine, so we must clarify a few points before starting our comparison. The basic premise of the M^3 project is that fast, semiconductor memory will soon be inexpensive enough so that many important databases (e.g., dozens of gigabytes) will fit inside main memory. When this occurs, it may be more cost effective to build a conventional machine with a massive memory, rather than building a machine with parallel search elements but with insufficient memory to hold the entire database. Thus, in our comparisons we will assume that the database fits within the M^3 memory, but it does not fit in a machine where resources were invested in parallel processing elements. The memory size / processor speed tradeoffs are discussed in detail in [3].

The query times in [1,2] are divided into query processing (or compiling) time, the actual database search time, and the time to transmit the answer back to a host machine. In this note we only study the database search time because the other times will be roughly the same in M^3 and other database machines. Furthermore, we compare the M^3 only to the NON-VON [2], the fastest of the database machines.

The M^3 processor speed plays a very important role in the evaluation. To be conservative, we assume that the M^3 has a 1 MIPS processor. However, at the end of this note we briefly consider the effect of a 10 MIPS processor, noting that this value is still very reasonable.

3.1. Query #1

This query is a select over a relation with 1,110 tuples. Each tuple is 127 bytes long. The search key for the select is 12 bytes long. The answer consists of 3 tuples, but only 21 bytes of each one are required for the answer. The NON-VON search time for this query is between 0.0827 (best case, data on disk) and 0.1067 (worst case, data also on disk) seconds. (Using the parameters of [2],

this is OVIO + BCOM + DAVAC + DROT.)

For M^3, the search time depends on the data structures available for the relation. For a sequential scan, we must examine each of the tuples. Assuming that it takes 10 machine instructions to examine a tuple (the key is 12 bytes or 4 words), this will take 0.011 seconds on a 1 MIPS machine. If a binary tree exists for the relation (and one of the premises of M^3 is that there will be enough memory to hold auxiliary structures for the important search fields), the time can be reduced considerably. The search would involve going down the tree (11 levels maximum and 10 instructions to examine each node), and extracting pointers to the three matching records (20 instructions), for a total of 130 microseconds. If a hash table exists, we would simply need to hash on the key and extract the pointers. Assuming 10 instructions per pointer, this would take 30 microseconds.

In summary, comparing against the best NON-VON times, the M^3 could provide anywhere from a 7 fold speedup (sequential search for M^3) to a 2700 fold speedup (hash table lookup for M^3). It is interesting to note that if we assume that NON-VON has all of the data in memory (which may not be fair since we are giving NON-VON both a large memory and parallel search elements), it still does not beat an M^3 that uses hashing. In this case, both search times are comparable (40 microseconds for NON-VON; 30 for M^3).

3.2. Query #2

The second query is a select of one relation, followed by a join of the result with a second relation. The first relation contains 282 (52-byte) tuples. The selection yields 22 tuples. The second relation contains 11,436 (127-byte) tuples. The join field is 20 bytes long, and 422 tuples are produced by the join. The NON-VON search times for this query are 0.336 (best case, data on disk) to 0.4667 (worst case, data also on disk) seconds.

The M^3 search times again depend on the data structures available. If none are available, we must first scan the first relation (282 tuples at 10 instructions per tuple). For each of the matching 22 tuples, we must set up a sequential scan of the second relation (20 instructions, say), and then scan (11,436 tuples at 15 instructions each). (Each check takes 15 and not 10 instructions as we had

assumed earlier because the join field is longer.) For each of the resulting 422 tuples, suppose we perform 10 additional instructions. Adding this up we obtain about 3.8 million instructions, or 3.8 seconds on a 1 MIPS machine.

However, if we construct a hash table to aid in the join we can reduce this time considerably. If we assume it takes 20 instructions to insert the key of each tuple of the second relation into a hash table, then 228,720 instructions will build the table. To check if each of the 22 keys resulting from the select exist in the table takes only 22 times say 20 instructions. As before, we include 2820 instructions to do the initial select, plus 4220 instructions to process the resulting tuples. This gives us a total search time of 0.24 seconds.

If search structures already exist for the second relation, then of course the time can be further reduced. For example, if a binary tree exists for the join field, the join involves looking up 22 keys (14 levels of the tree times 15 instructions at each node). Adding this to the select time and the time to process the 422 results, we get a total search time of 0.012 seconds.

In summary, without auxiliary data structures M^3 will be about 11 times slower than NON-VON (best time). However, if M^3 is allowed to build its data structures, it can be 1.4 times faster than NON-VON. If the structures are already in place, the speedup is greater: 28 times.

3.3. Query #3

The last query examines a relation with 194 (256-byte) tuples. The values in a given field (encumb, 4 bytes) are to be added for each group of tuples that match in a second field (acct-fund, 8 bytes). There are 17 unique values of the acct-fund fields. The NON-VON search times are 0.088 (best case, data on disk) and 0.11 (worst case) seconds.

On an M^3 we would always have to scan the entire relation, i.e., 194 tuples times say 10 instructions per tuple. The results can be collected by building a linked list, where each element contains the current sum for a given acct-fund value. To add each new value, we must scan the list to find the proper record. Since there will be at most 17 records, a scan will take on the average 9 records, at say 10 instructions each. Thus, each insertion takes 90 instructions, and this

must be multiplied by the 194 tuples that exist. The total time is then 0.019 seconds.

Since there are so few records in the linked list of partial sums, changing this data structure does not bring large improvements. For example, with a B-tree (5 levels maximum), each insertion will take roughly 50 instructions, for a total time of 0.012 seconds.

Comparing these numbers to the NON-VON times, we see that M^3 is a factor of 4 to 7 times faster on this query.

3.4. Conclusions

Our rough estimates clearly indicate that M^3 can provide significant speedups for the sample queries of [1,2]. To summarize the results, we present the following table that gives the M^3 speedup (i.e., the NON-VON search time divided by the M^3 search time) for the case where search structures and data are available in M^3 memory, and data is on disk in NON-VON. We also give the speedup attainable if the M^3 processor ran at *10 MIPS*.

M^3 Speedup		
	1 MIPS Processor	10 MIPS Processor
Query #1	2,700	27,000
Query #2	28	280
Query #3	7	70

As Hillyer, Shaw, and Nigam [2] state, "There are hazards in attempting to deduce the relative merit of alternative architectures based on 'paper and pencil' analysis of performance on a small number of specific problems with specified data." We certainly agree with them: the results we have presented must be treated with caution. However, we do feel that they illustrate that memory can be an extremely useful resource and can provide impressive speedups, even when

the competition is a powerful database machine like NON-VON.

References

[1] P. B. Hawthorn and D. J. DeWitt, "Performance Analysis of Alternative Database Machine Architectures," *IEEE Transactions on Software Engineering*, Vol. SE-8, Num. 1, January 1982.

[2] B. K. Hillyer, D. E. Shaw, and A. Nigam, "NON-VON's Performance on Certain Database Benchmarks," Unpublished Technical Report, Columbia University, 1984.

[3] H. Garcia-Molina, R. Cullingford, P. Honeyman, and R. Lipton, "The Case for Massive Memory," Unpublished Technical Report, Princeton University, 1984.

END

FILMED

2-85

DTIC